Lecture 10:

Directory-Based Coherence + Implementing Synchronization

Parallel Computing

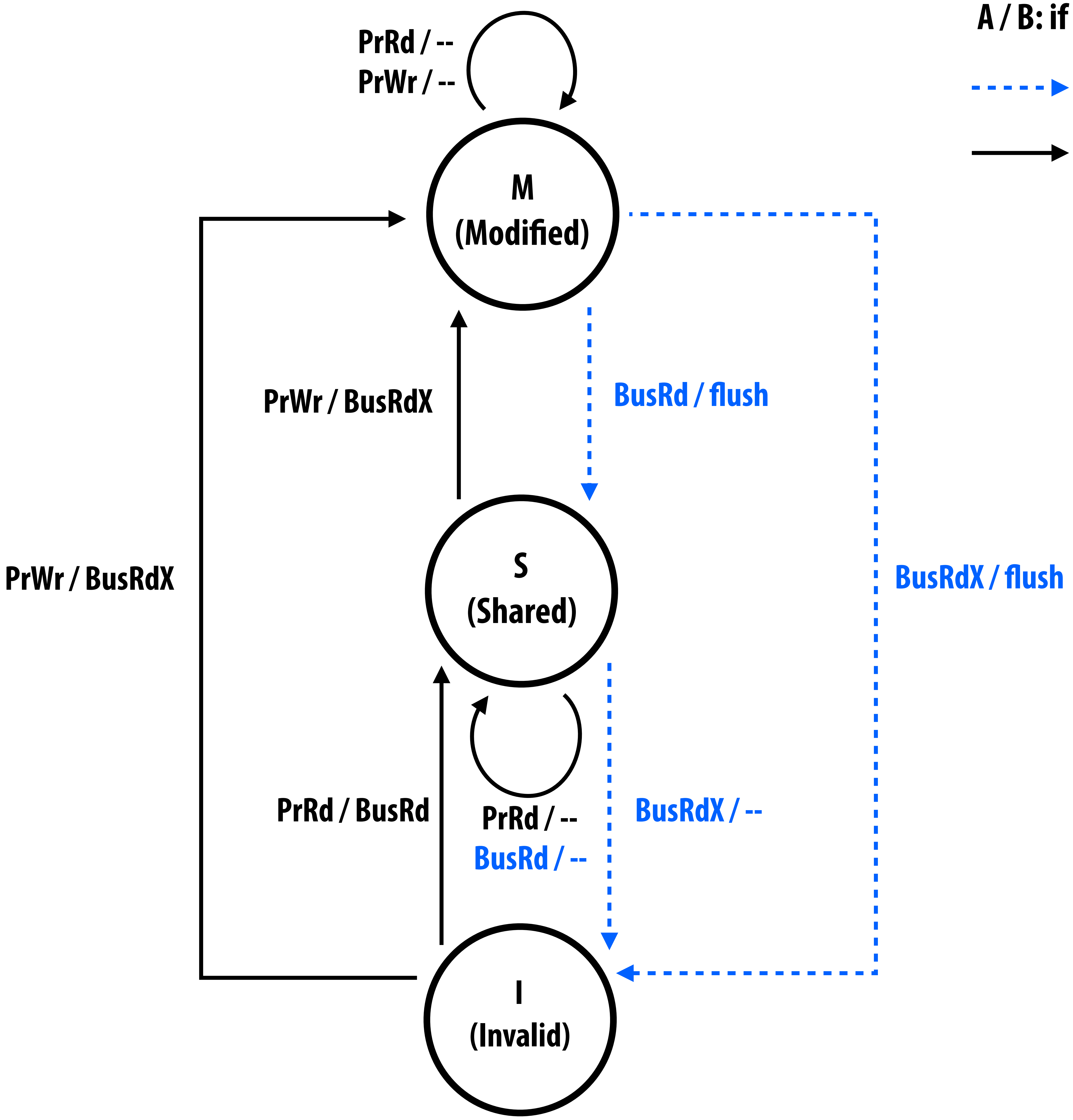
Stanford CS149, Winter 2019

Today's topics

- A quick discussion of directory-based cache coherence

- Efficiently implementing synchronization primitives
    - Primitives for ensuring mutual exclusion
        - Locks
        - Atomic primitives (e.g., atomic_add)
        - Transactions (later in the course)
    - Primitives for event signaling
        - Barriers

Review: MSI state transition diagram *



A / B: if action A is observed by cache controller, action B is taken

- - - -▶ Remote processor (coherence) initiated transaction

——▶ Local processor initiated transaction

flush = flush dirty line to memory

* Remember, all caches are carrying out this logic independently to maintain coherence

Example

P0: LD X

P0: LD X

P0: ST X ← 1

P0: ST X ← 2

P1: ST X ← 3

P1: LD X

P0: LD X

P0: ST X ← 4

P1: LD X

P0: LD Y

P0: ST Y ← 1

P1: ST Y ← 2

Consider this sequence of loads and stores to addresses X and Y by processors P0 and P1

Assume that X and Y contain value 0 at start of execution.

Directory-based cache coherence

What you should know

- What limits the scalability of snooping-based approaches to cache coherence?

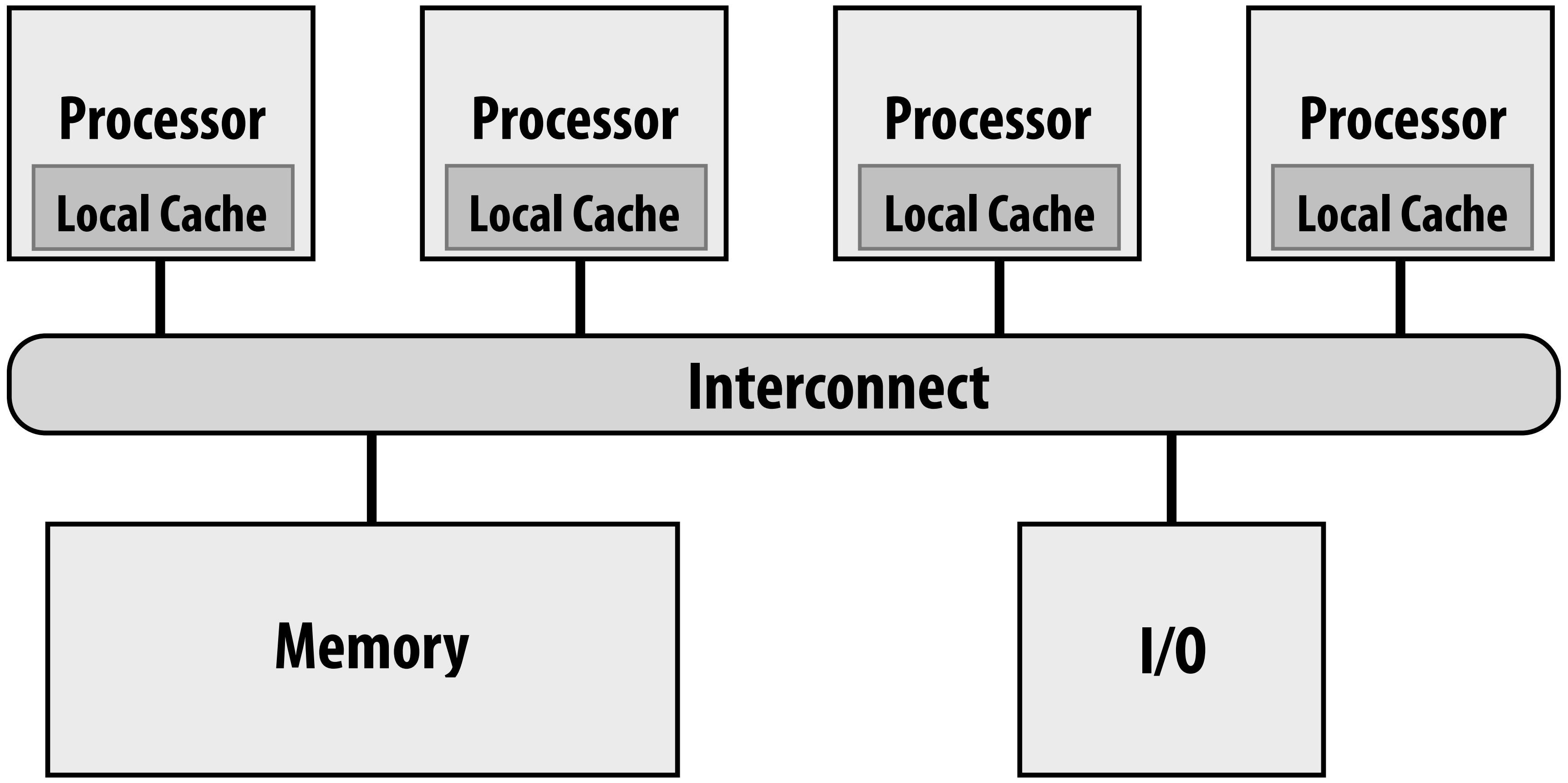
- How does a directory-based scheme avoid these problems?

- How can the storage overhead of the directory structure be reduced? (and at what cost?)

Implementing cache coherence

The snooping cache coherence protocols discussed last week relied on broadcasting coherence information to all processors over the chip interconnect.

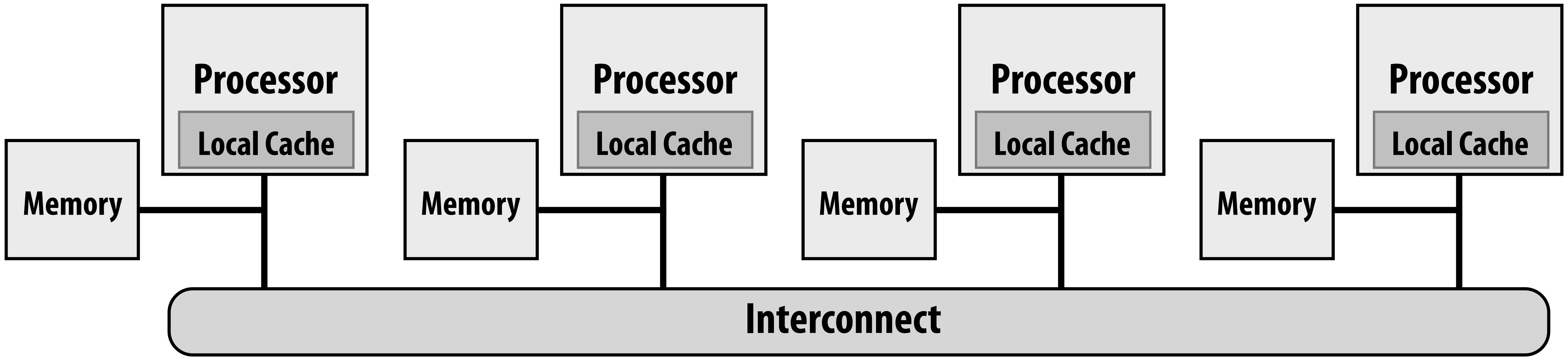
Every time a cache miss occurred, the triggering cache communicated with all other caches!



We discussed what information was communicated and what actions were taken to implement the coherence protocol.

We did not discuss how to implement broadcasts on an interconnect.
(one example is to use a shared bus for the interconnect)

Problem: scaling cache coherence to large machines



Recall idea of non-uniform access shared memory systems (NUMA): locating regions of memory near the processors increases scalability: it yields higher aggregate bandwidth and reduced latency (especially when there is locality in the application)

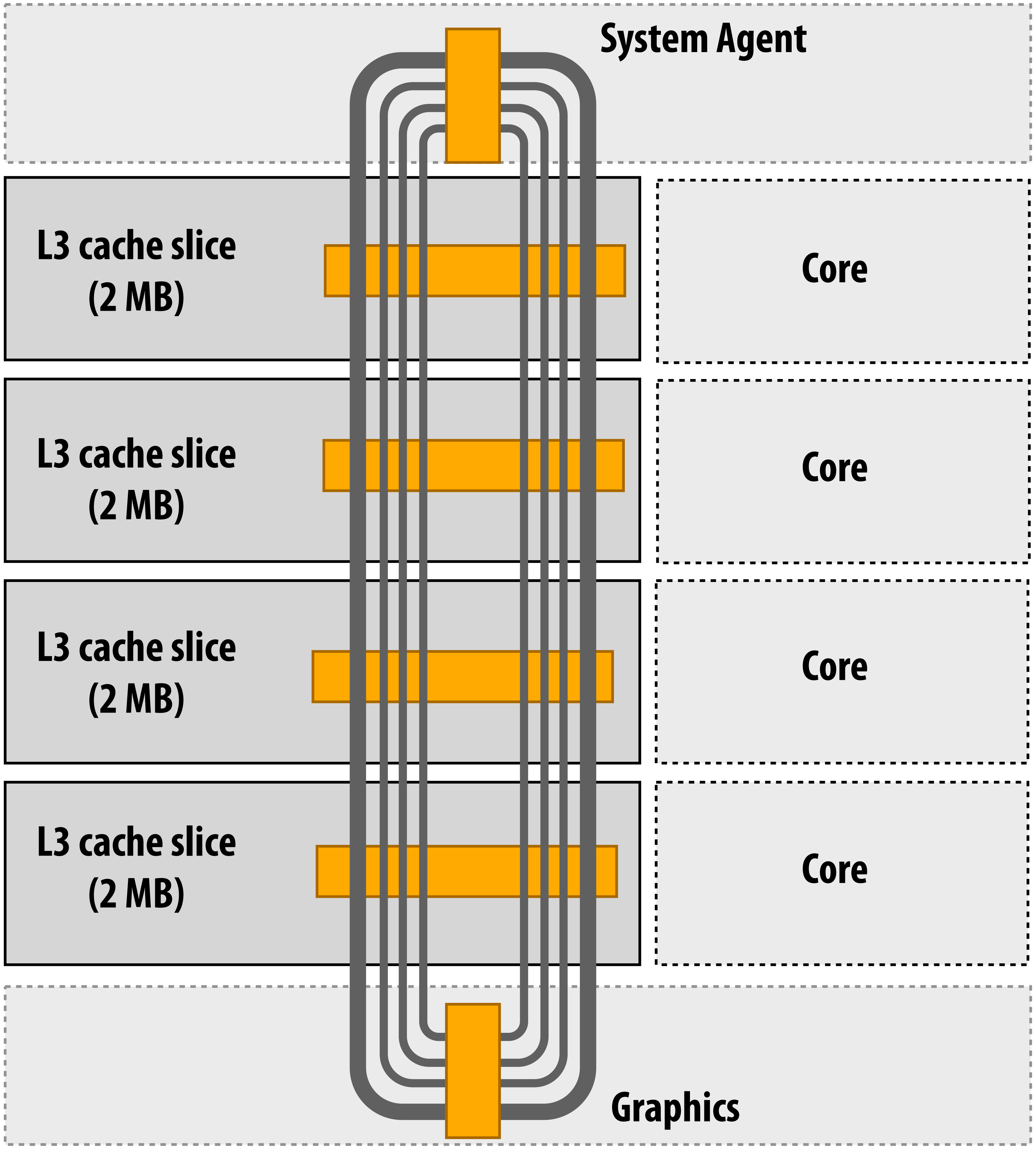
But... efficiency of NUMA system does little good if the coherence protocol can't also be scaled!

Consider this case: processor accesses nearby memory (good...), but to ensure coherence still must broadcast to all other processors it is doing so (bad...)

Some common terminology:

- cc-NUMA = "cache-coherent, non-uniform memory access"
- Distributed shared memory system (DSM): cache coherent, shared address space, but architecture implemented by physically distributed memories

Intel's ring interconnect



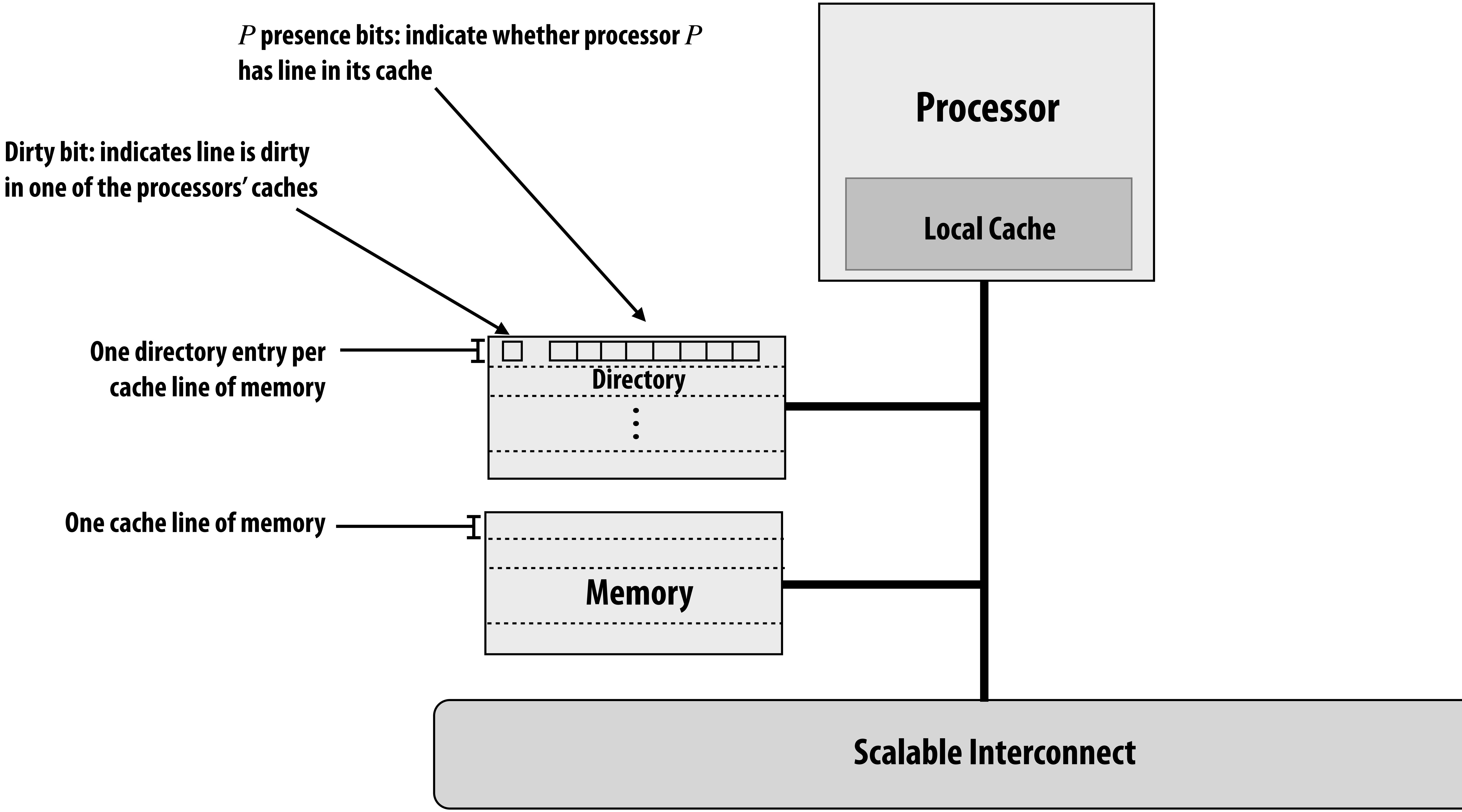
- **Multiple rings**
  - request
  - snoop
  - ack
  - data (32 bytes)
- **Six interconnect nodes: four "slices" of L3 cache + system agent + graphics**
- **Each bank of L3 connected to ring bus twice**

Scalable cache coherence using directories

- Snooping schemes broadcast coherence messages to determine the state of a line in the other caches

- Alternative idea: avoid broadcast by storing information about the status of the line in one place: a “directory”
  - The directory entry for a cache line contains information about the state of the cache line in all caches.
  - Caches look up information from the directory as necessary
  - Cache coherence is maintained by point-to-point messages between the caches on a “need to know” basis (not by broadcast mechanisms)

A very simple directory



P presence bits: indicate whether processor P has line in its cache

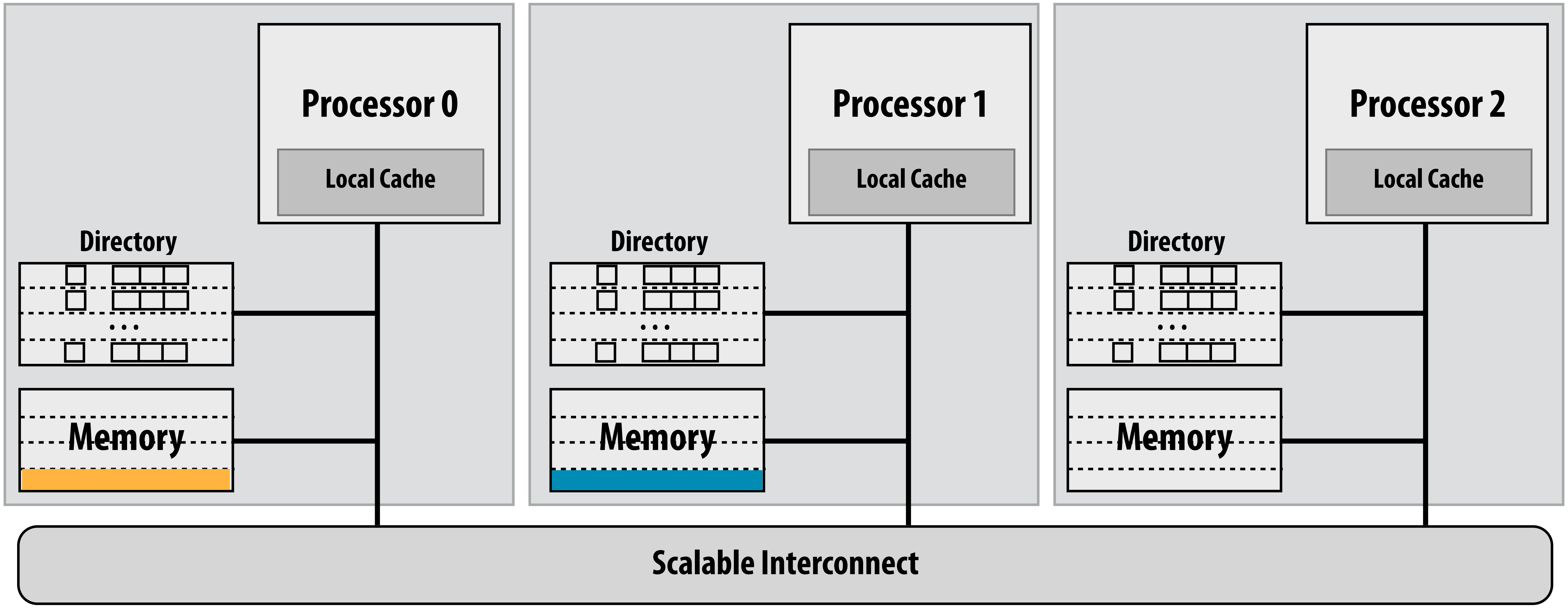
Dirty bit: indicates line is dirty in one of the processors' caches

One directory entry per cache line of memory

One cache line of memory

A distributed directory

Example: directory partition is co-located with memory it describes

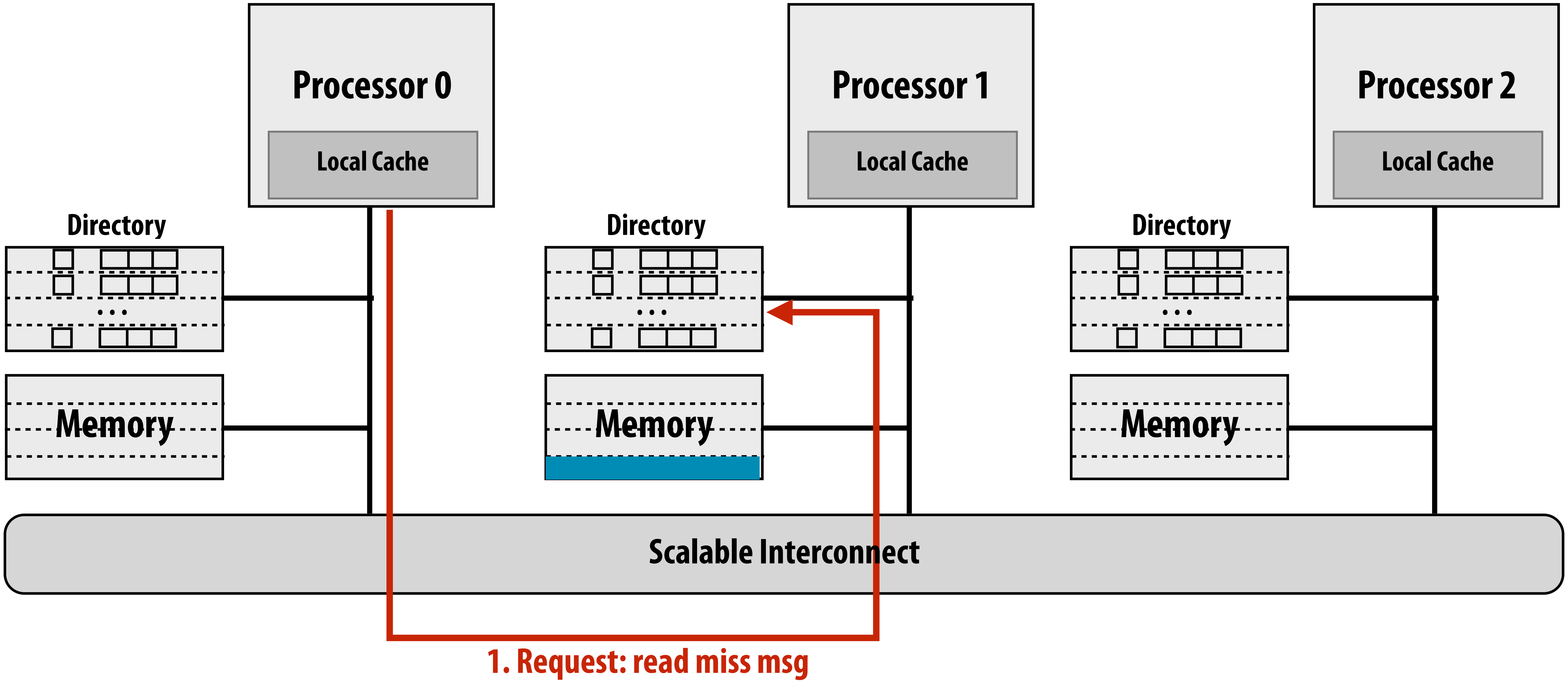


- “Home node” of a line: node with memory holding the corresponding data for the line
  Example: node 0 is the home node of the yellow line, node 1 is the home node of the blue line

- “Requesting node”: node containing processor requesting line

Example 1: read miss to clean line

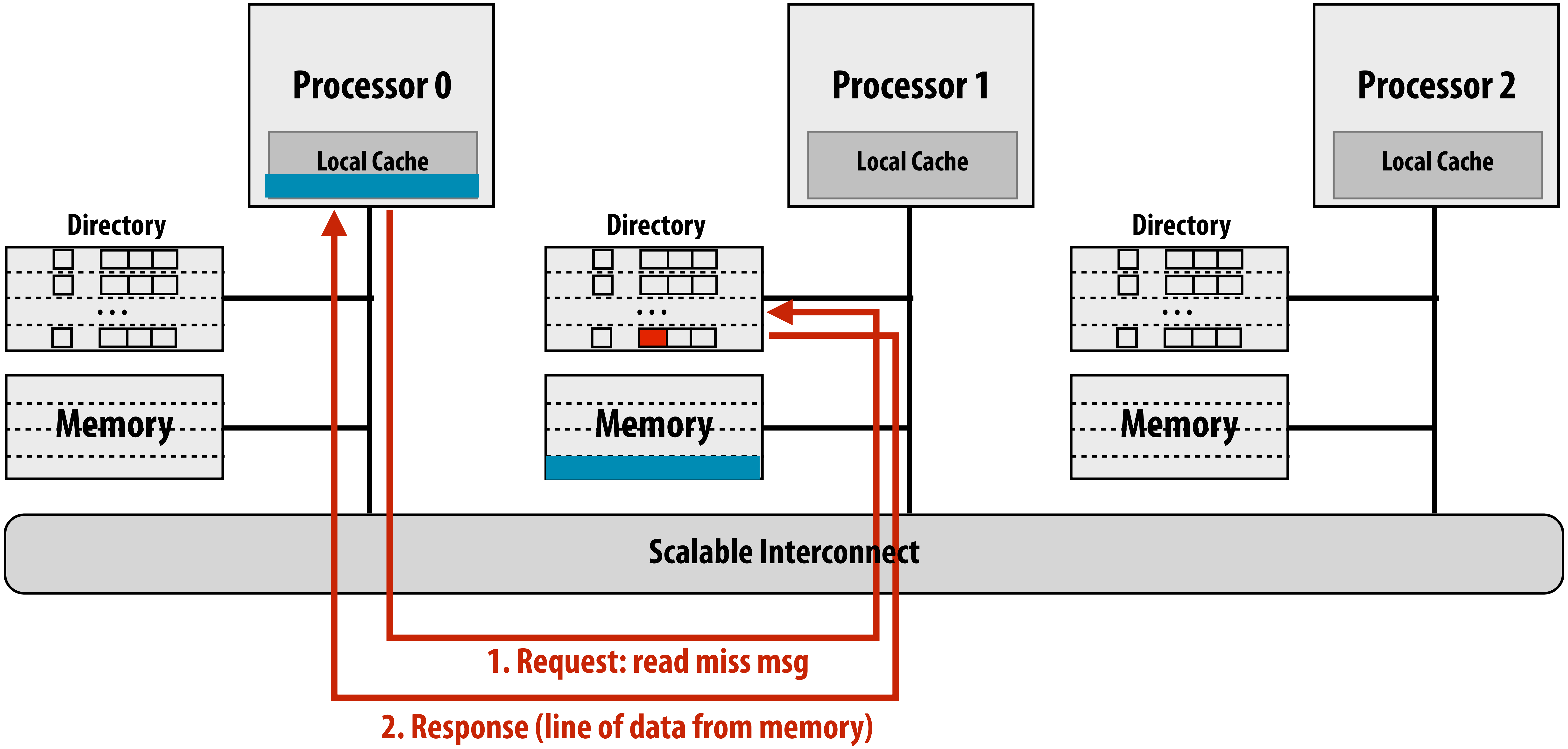
Read from main memory by processor 0 of the blue line: line is not dirty



- Read miss message sent to home node of the requested line
- Home directory checks entry for line

Example 1: read miss to clean line

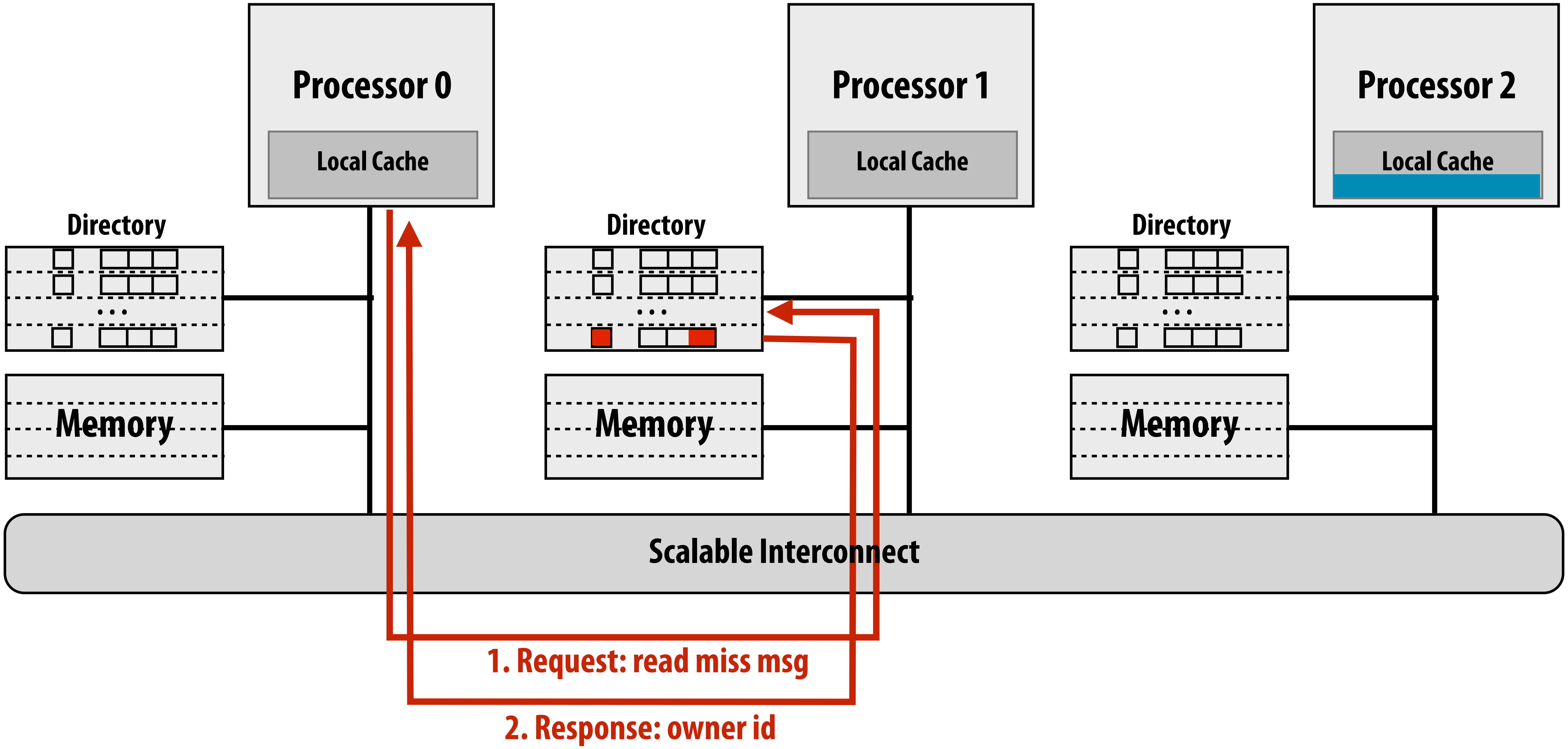
Read from main memory by processor 0 of the blue line: line is not dirty



- Read miss message sent to home node of the requested line
- Home directory checks entry for line
  - If dirty bit for cache line is OFF, respond with contents from memory, set presence[0] to true (to indicate line is cached by processor 0)

Example 2: read miss to dirty line

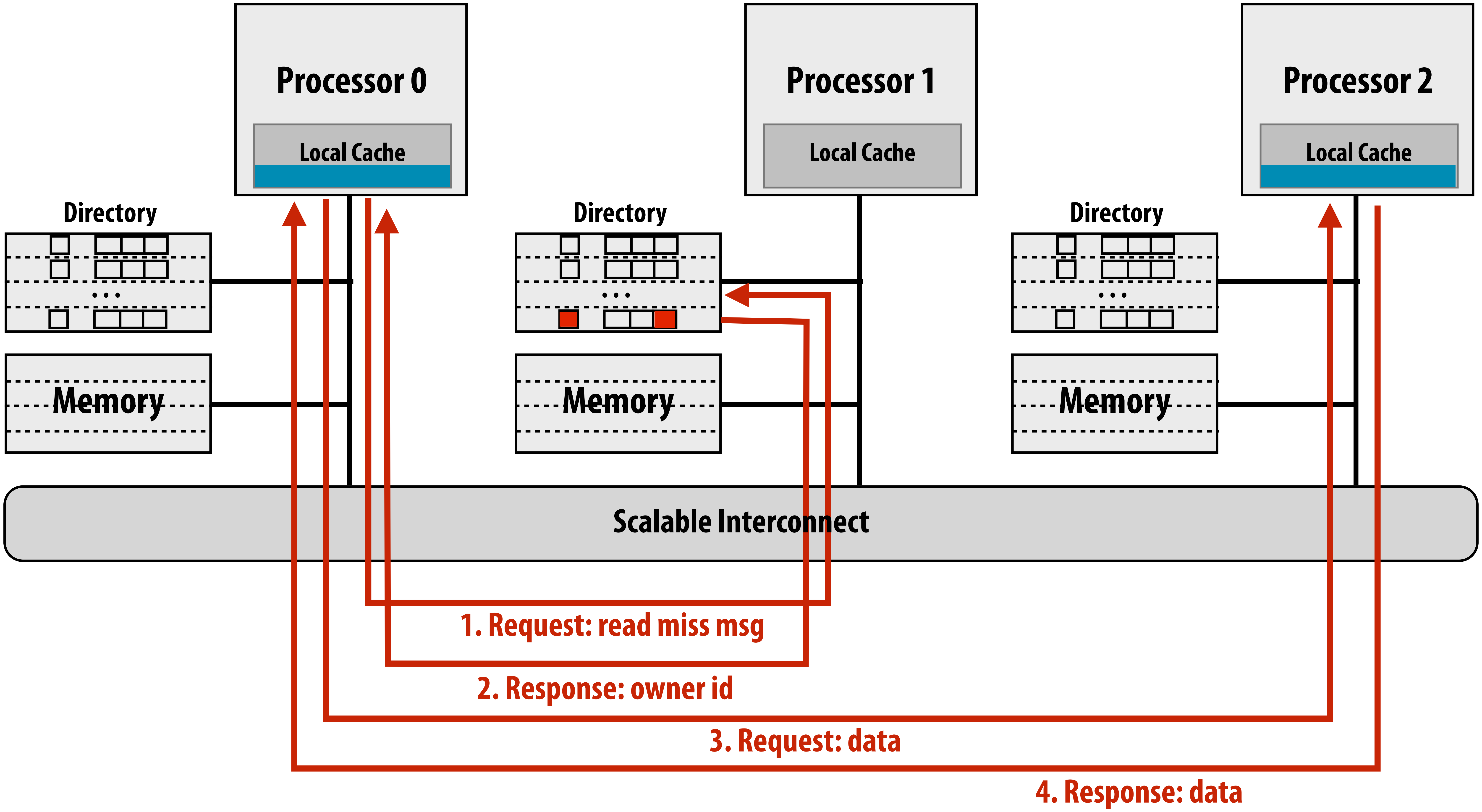
Read from main memory by processor 0 of the blue line: line is dirty (contents in P2's cache)



- If dirty bit is ON, then data must be sourced by another processor (with the most up-to-date copy of the line)
- Home node must tell requesting node where to find data
  - Responds with message providing identity of line owner ("get it from P2")

Example 2: read miss to dirty line

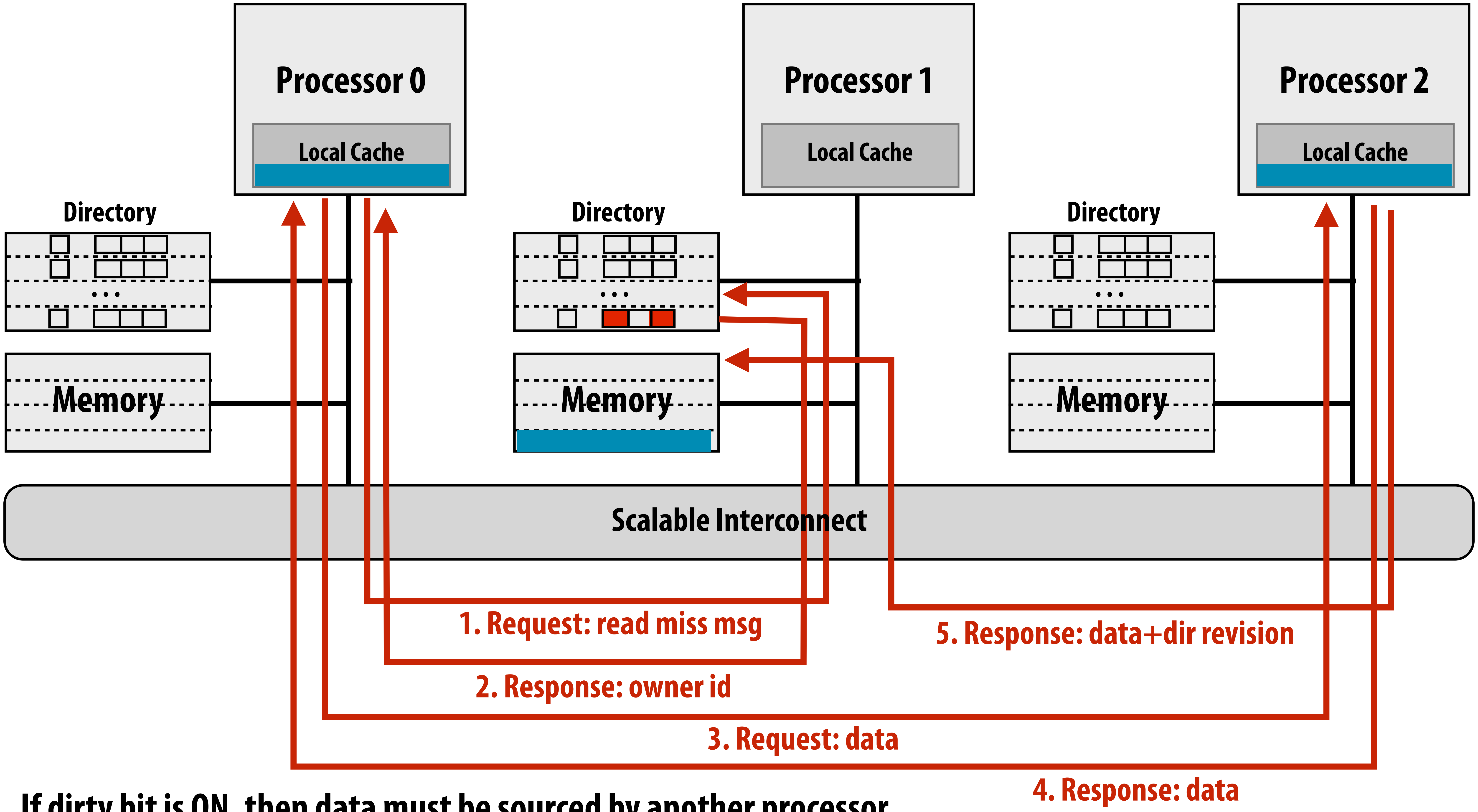
Read from main memory by processor 0 of the blue line: line is dirty (contents in P2's cache)



1. If dirty bit is ON, then data must be sourced by another processor
2. Home node responds with message providing identity of line owner
3. Requesting node requests data from owner
4. Owner changes state in cache to SHARED (read only), responds to requesting node

Example 2: read miss to dirty line

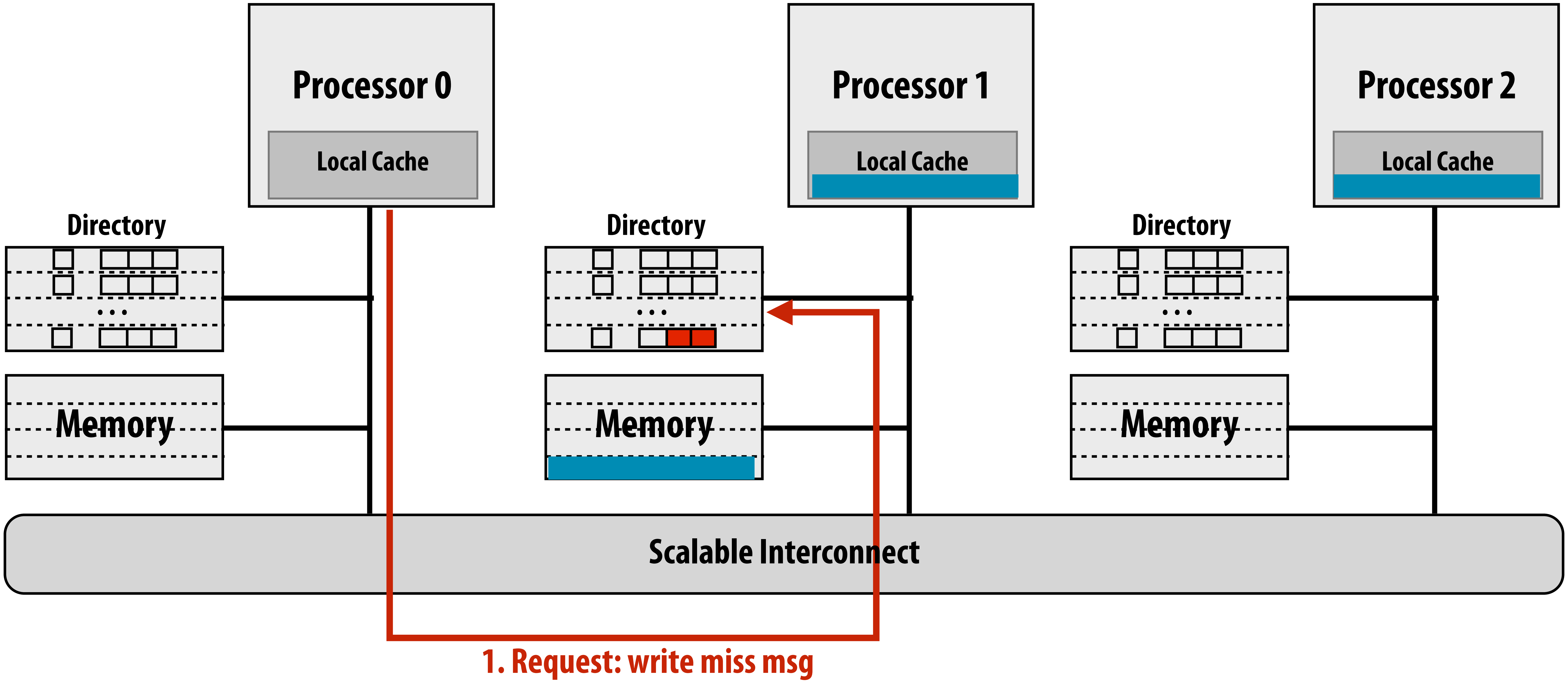
Read from main memory by processor 0 of the blue line: line is dirty (contents in P2's cache)



1. If dirty bit is ON, then data must be sourced by another processor
2. Home node responds with message providing identity of line owner
3. Requesting node requests data from owner
4. Owner responds to requesting node, changes state in cache to SHARED (read only)
5. Owner also responds to home node, home clears dirty, updates presence bits, updates memory

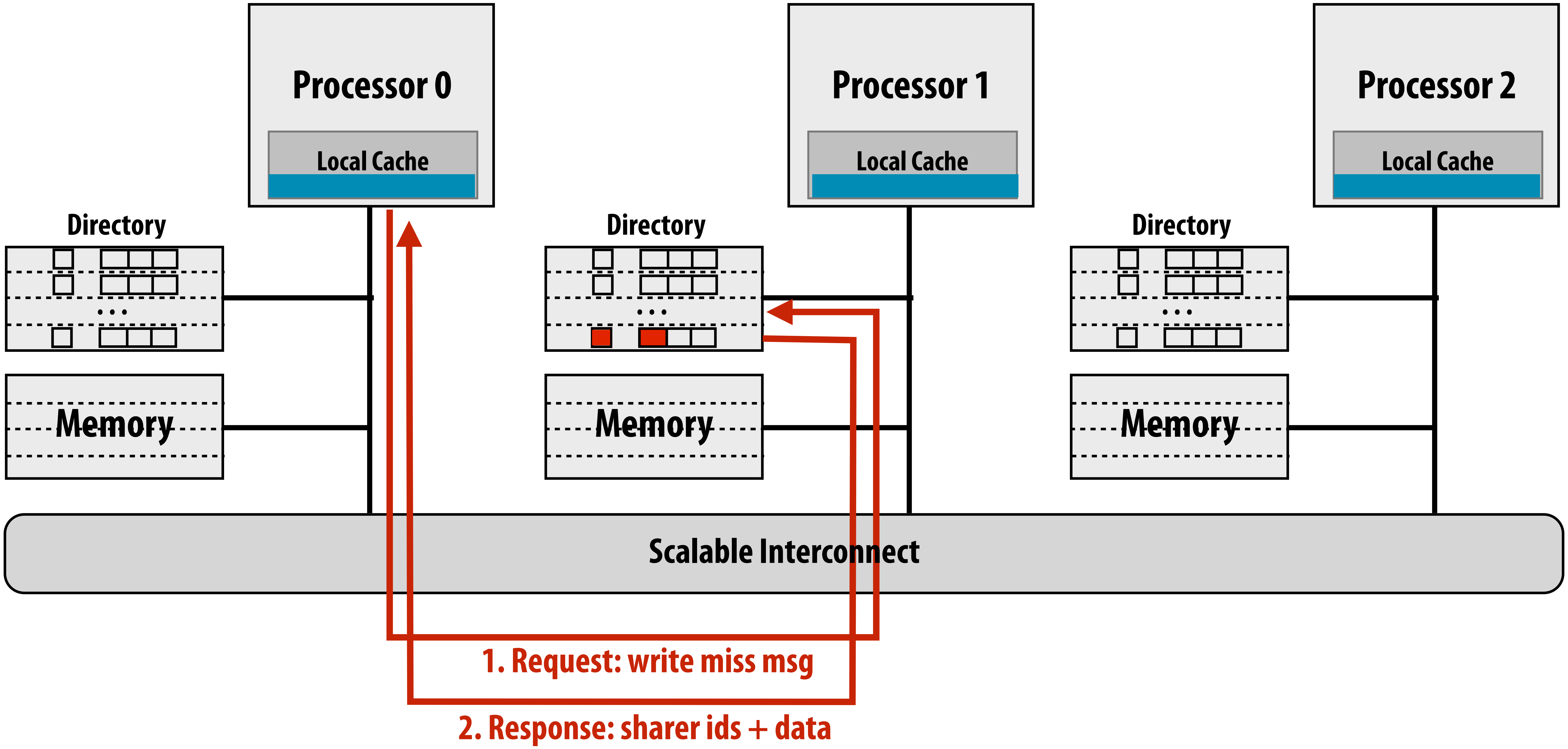
Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches



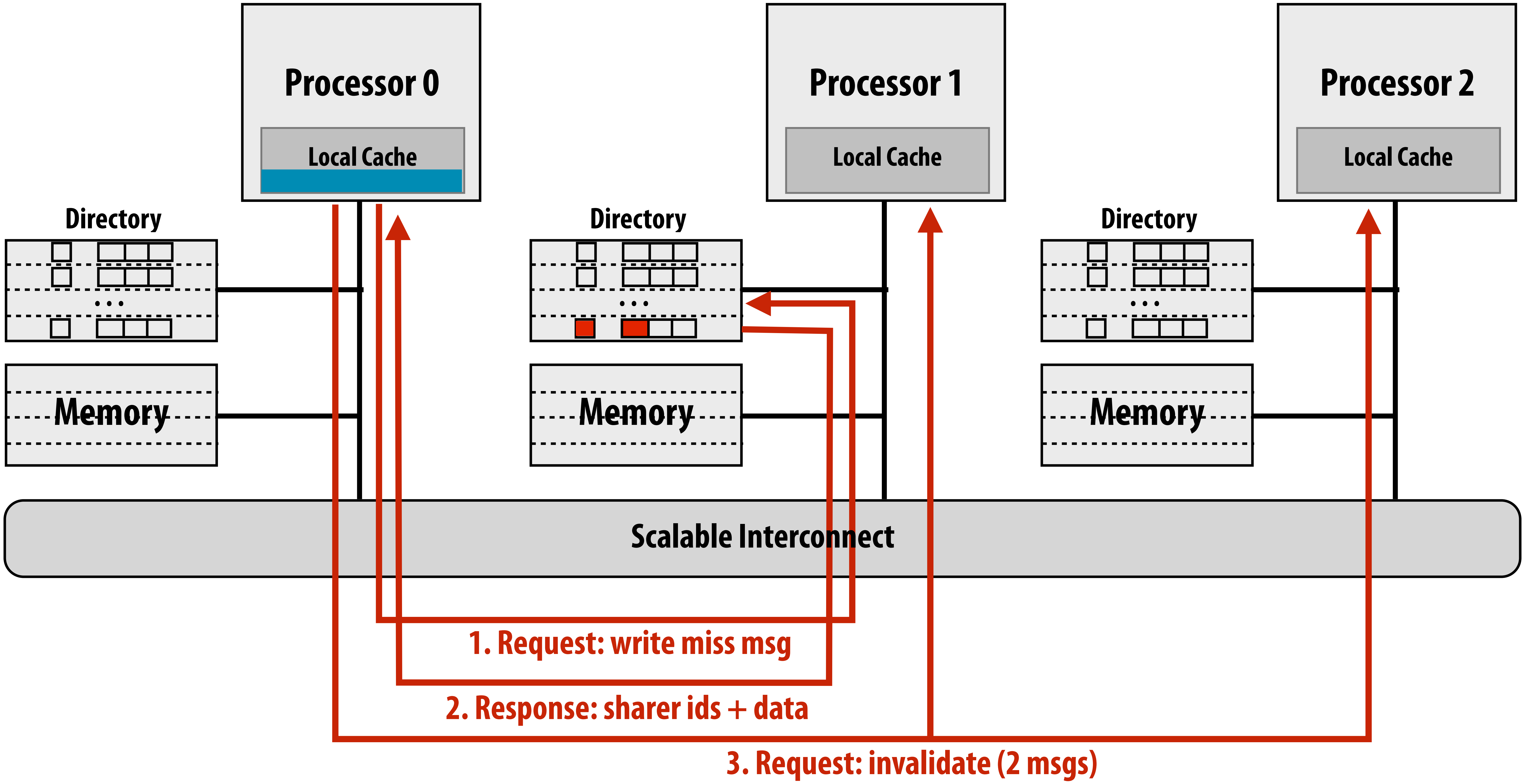
Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches



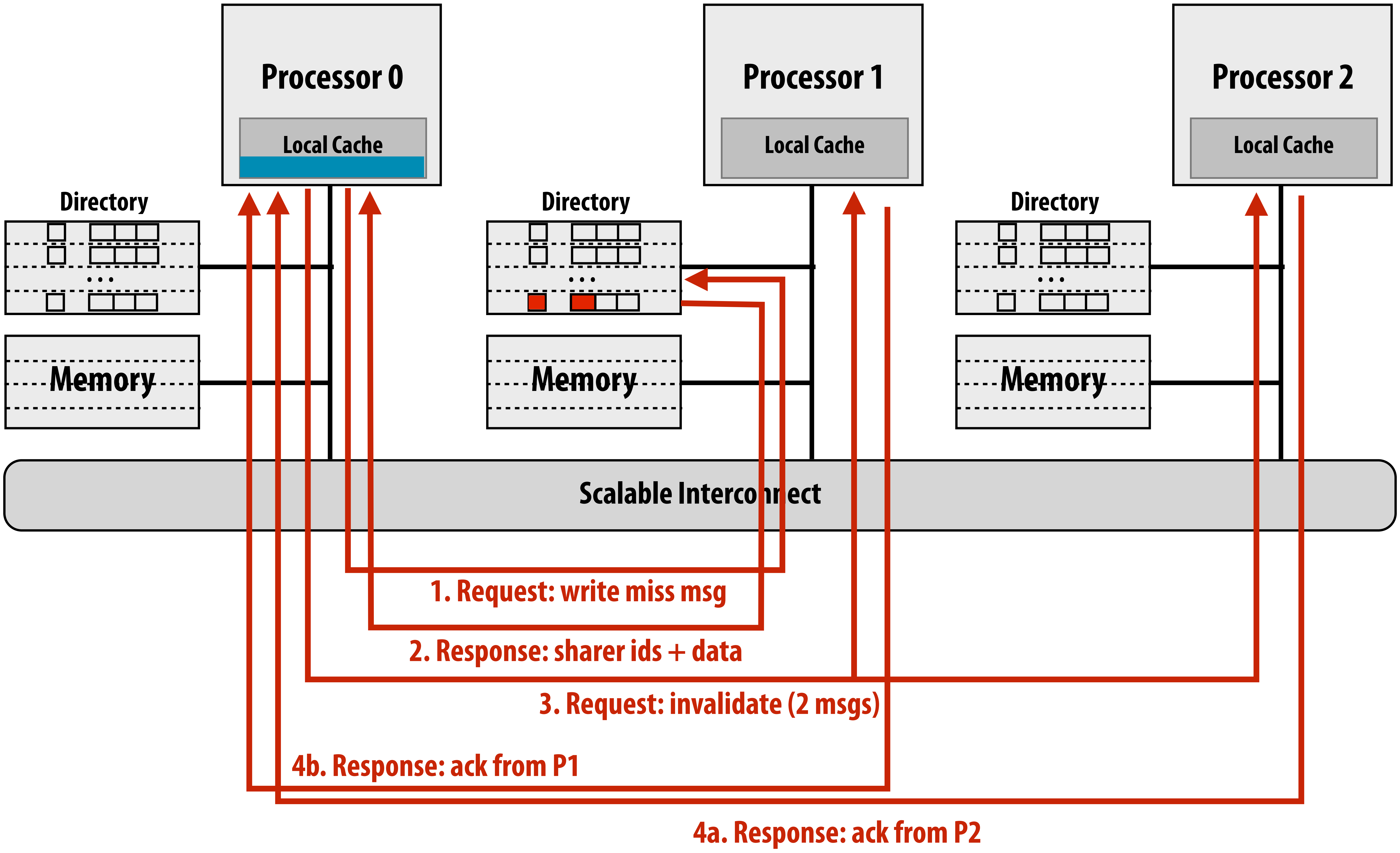
Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches



Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches



Processor 0
Local Cache
Processor 1
Local Cache
Processor 2
Local Cache
Directory
Directory
Directory
Memory
Memory
Memory
Scalable Interconnect
1. Request: write miss msg
2. Response: sharer ids + data
3. Request: invalidate (2 msgs)
4b. Response: ack from P1
4a. Response: ack from P2

After receiving both invalidation acks, P0 can perform write

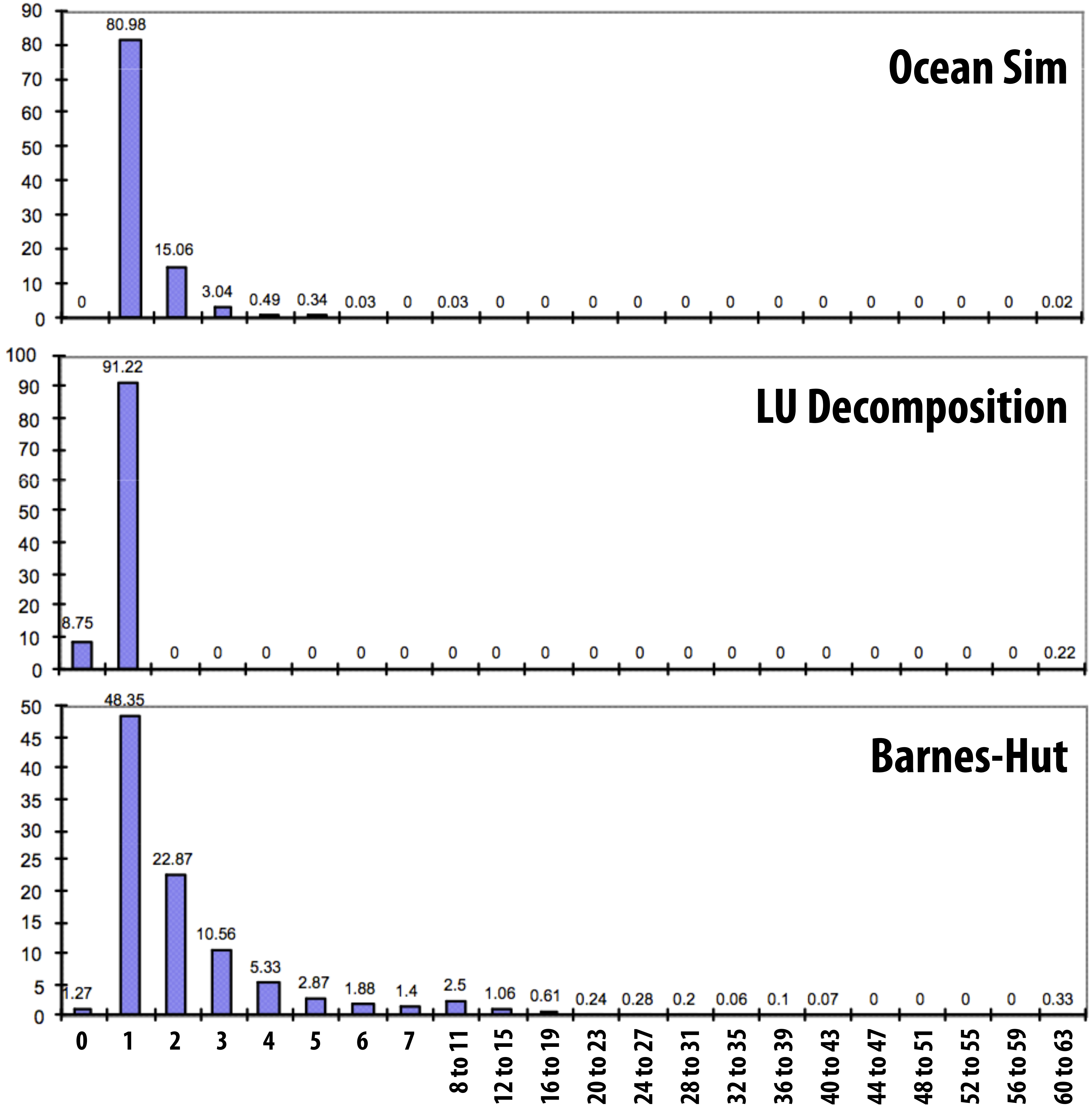
Advantage of directories

- **On reads, directory tells requesting node exactly where to get the line from**
    - Either from home node (if the line is clean)
    - Or from the owning node (if the line is dirty)
    - Either way, retrieving data involves only point-to-point communication

- **On writes, the advantage of directories depends on the number of sharers**
    - In the limit, if all caches are sharing data, all caches must be communicated with (just like broadcast in a snooping protocol)

Cache invalidation patterns

64 processor system



Graphs plot histogram of number of sharers of a line at the time of a write

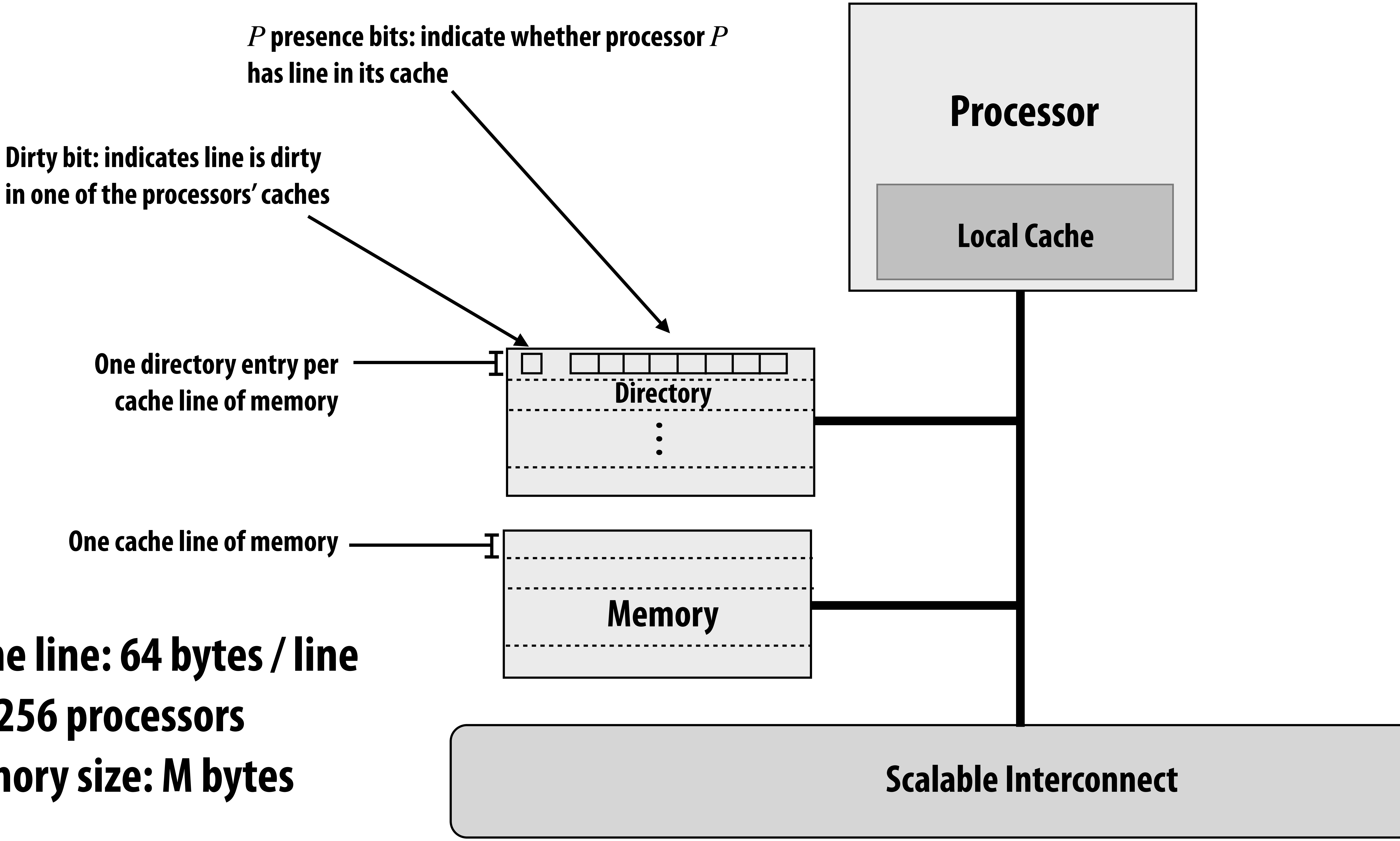
In general only a few processors share the line (only a few processors must be told of writes)

Not shown here, but the expected number of sharers typically increases slowly with P (good!)

In general, only a few sharers during a write

- **Access patterns**
  - "Mostly-read" objects: lots of sharers, but writes are infrequent, so communicating with all sharers on a write has minimal impact on performance
  - Migratory objects (one processor reads/writes for while, then another, etc.): very few sharers, so count does not scale with number of processors
  - Frequently read/written objects: frequent invalidations, but sharer count is low because count cannot build up in short time between invalidations (e.g, shared task queue)
  - Low-contention locks: infrequent invalidations, so no performance problem
  - High-contention locks: tricky because many readers present when lock released
- **Implication 1: directories are useful for limiting coherence traffic**
  - Don't need a broadcast mechanism to "tell everyone"
- **Implication 2: suggests ways to optimize directory implementations (reduce storage overhead)**

How big is the directory?


P presence bits: indicate whether processor P has line in its cache
Dirty bit: indicates line is dirty in one of the processors' caches
Processor
Local Cache
One directory entry per cache line of memory
Directory
One cache line of memory
Memory
Scalable Interconnect

Cache line: 64 bytes / line

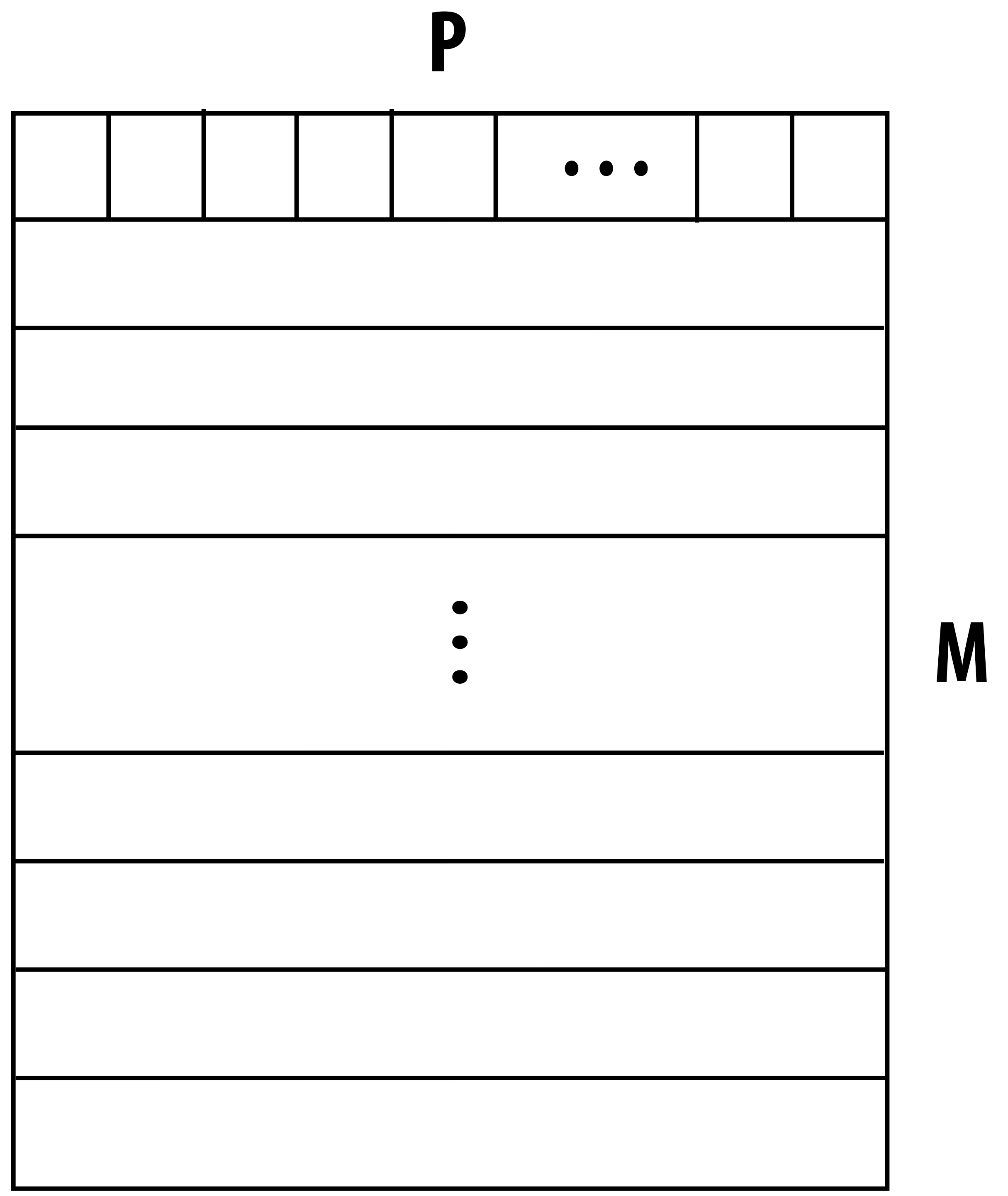
P = 256 processors

Memory size: M bytes

How big is the directory?

Full bit vector directory representation

- Recall: one presence bit per node
- Storage proportional to P x M
  - P = number of nodes (e.g., processors)
  - M = number of lines in memory
- Storage overhead rises with P
  - Assume 64 byte cache line size (512 bits)
  - 64 nodes (P=64) → 12% overhead
  - 256 nodes (P=256) → 50% overhead
  - 1024 nodes (P=1024) → 200% overhead



Reducing storage overhead of directory

- **Optimizations on full-bit vector scheme**
  - Increase cache line size (reduce M term)
    - What are possible problems with this approach? (consider graphs from last lecture)
  - Group multiple processors into a single directory "node" (reduce P term)
    - Need only one directory bit per node, not one bit per processor
    - Hierarchical: could use snooping protocol to maintain coherence among processors in a node, directory across nodes
- **We will now discuss one alternative scheme**
  - Limited pointer schemes (reduce P)

Limited pointer schemes

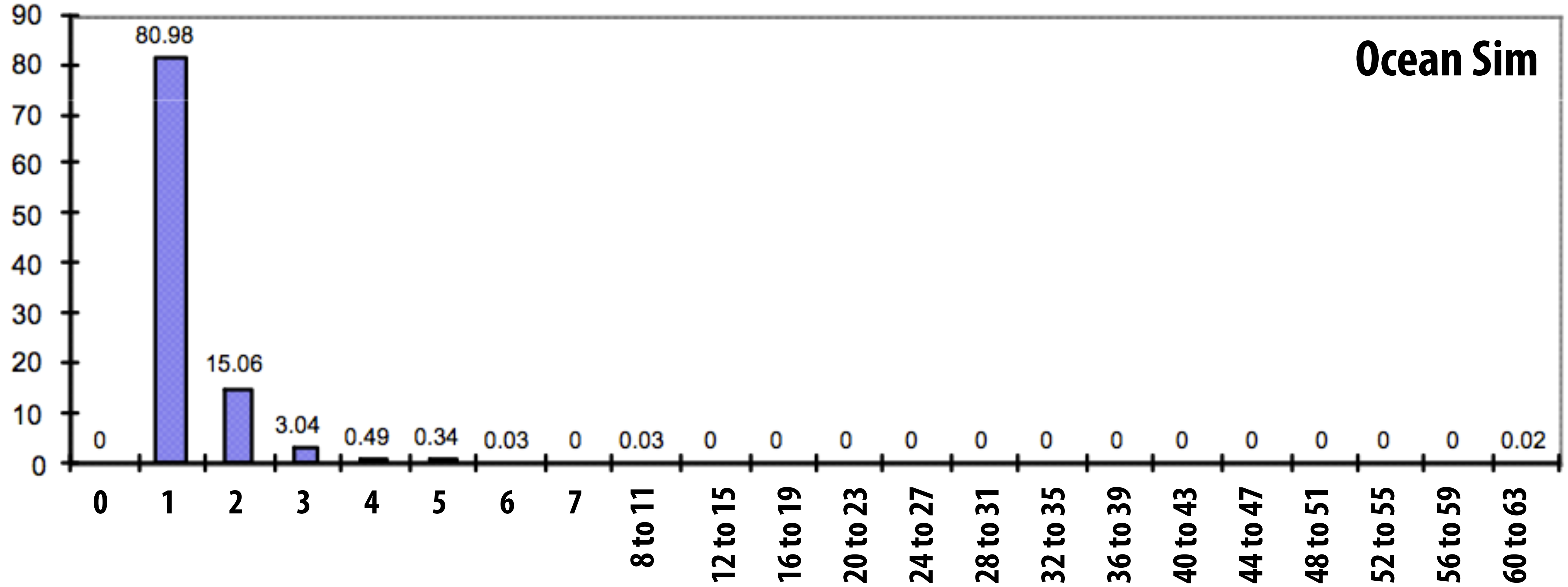
Since data is expected to only be in a few caches at once, storage for a limited number of pointers per directory entry should be sufficient (only need a list of the nodes holding a valid copy of the line!)

Example: 1024 processor system

Full bit vector scheme needs 1024 bits per line

Instead, can store ~100 pointers to nodes holding the line ($\log_2(1024)=10$ bits per pointer)

In practice, our workload evaluation says we can get by with far less than this



Managing overflow in limited pointer schemes

- Fallback to broadcast (if broadcast mechanism exists)
  - When more than max number of sharers, revert to broadcast

- If no broadcast mechanism is present on machine
  - Do not allow more than a max number of sharers
  - On overflow, newest sharer replaces an existing one (must invalidate line in the old sharer's cache)

- Coarse vector fallback
  - Revert to bit vector representation representation
  - Each bit corresponds to K nodes
  - On write, invalidate all nodes a bit corresponds to

Optimizing for the common case

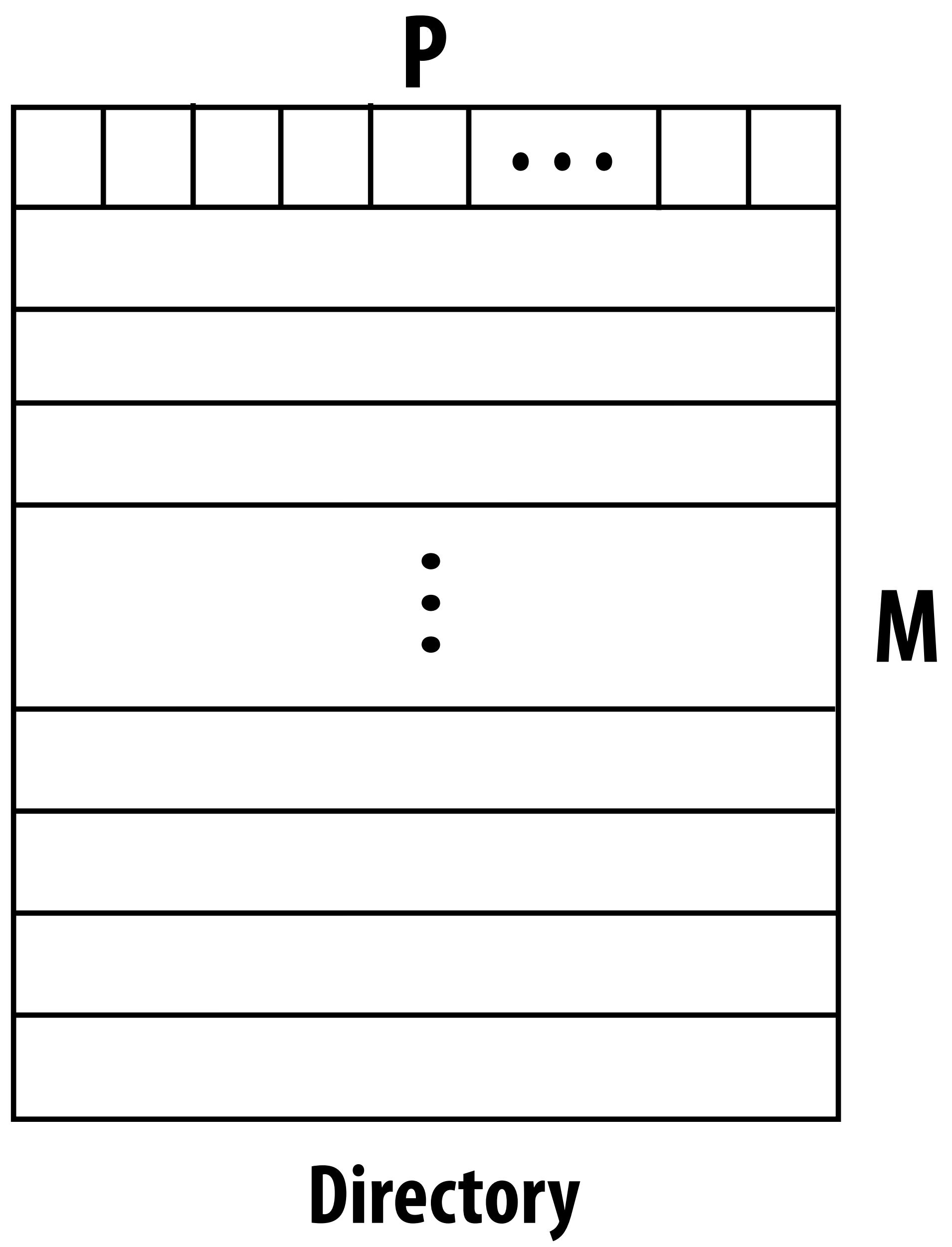
Limited pointer schemes are a great example of smartly understanding and optimizing for the common case:

1. **Workload-driven observation: in general the number of cache line sharers is low**
2. **Make the common case simple and fast: array of pointers for first N sharers**
3. **Uncommon case is still handled correctly, just with a slower, more complicated mechanism (the program still works!)**
4. **Extra expense of the complicated solution is tolerable, since it happens infrequently**

Limited pointer schemes: summary

- Limited pointer schemes reduce directory storage overhead caused by large P
  - By adopting a compact representation of a list of sharers

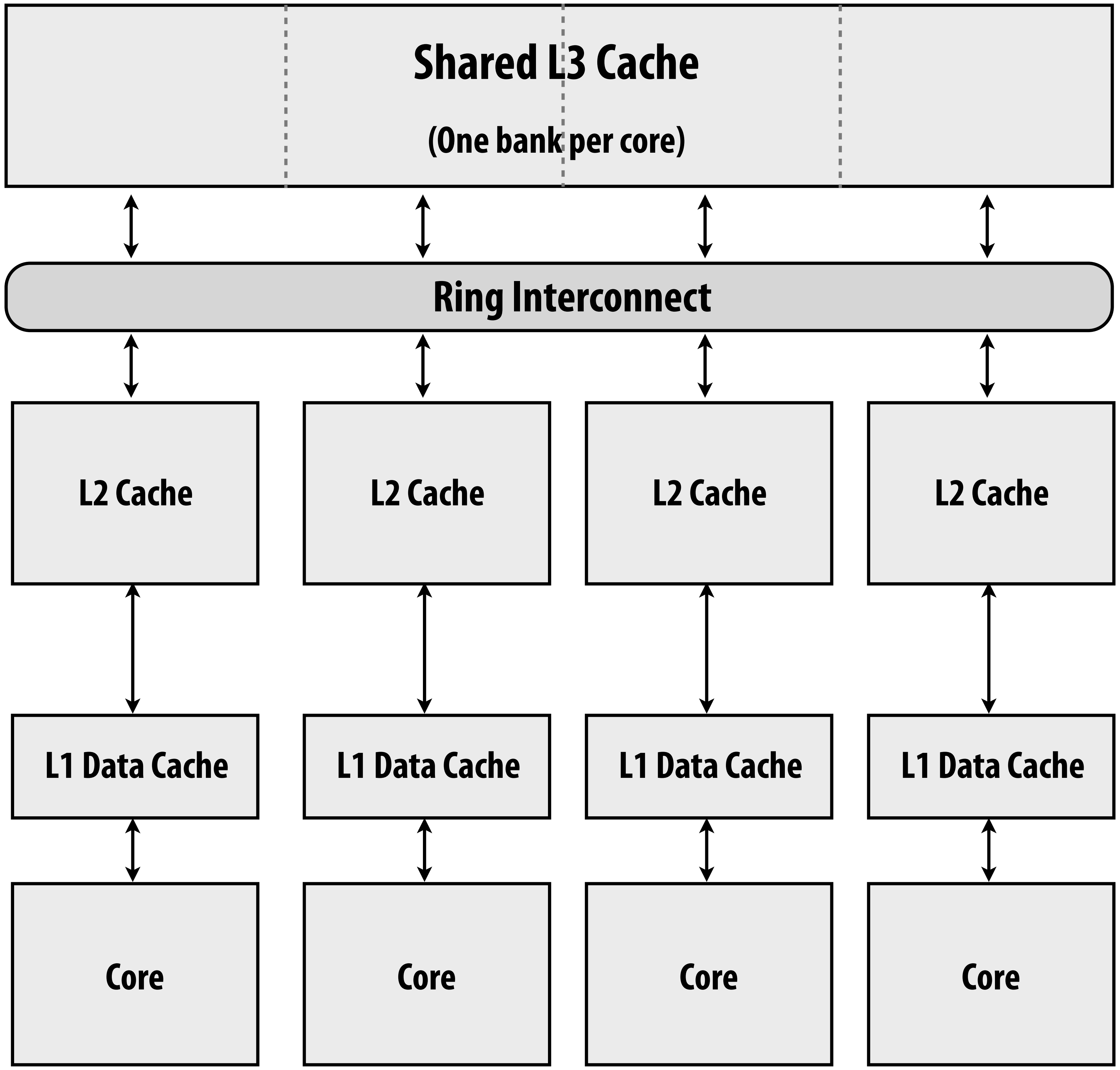
- But do we really even need to maintain a list of sharers for each cache-line-sized chunk of data in memory?



Limiting size of directory

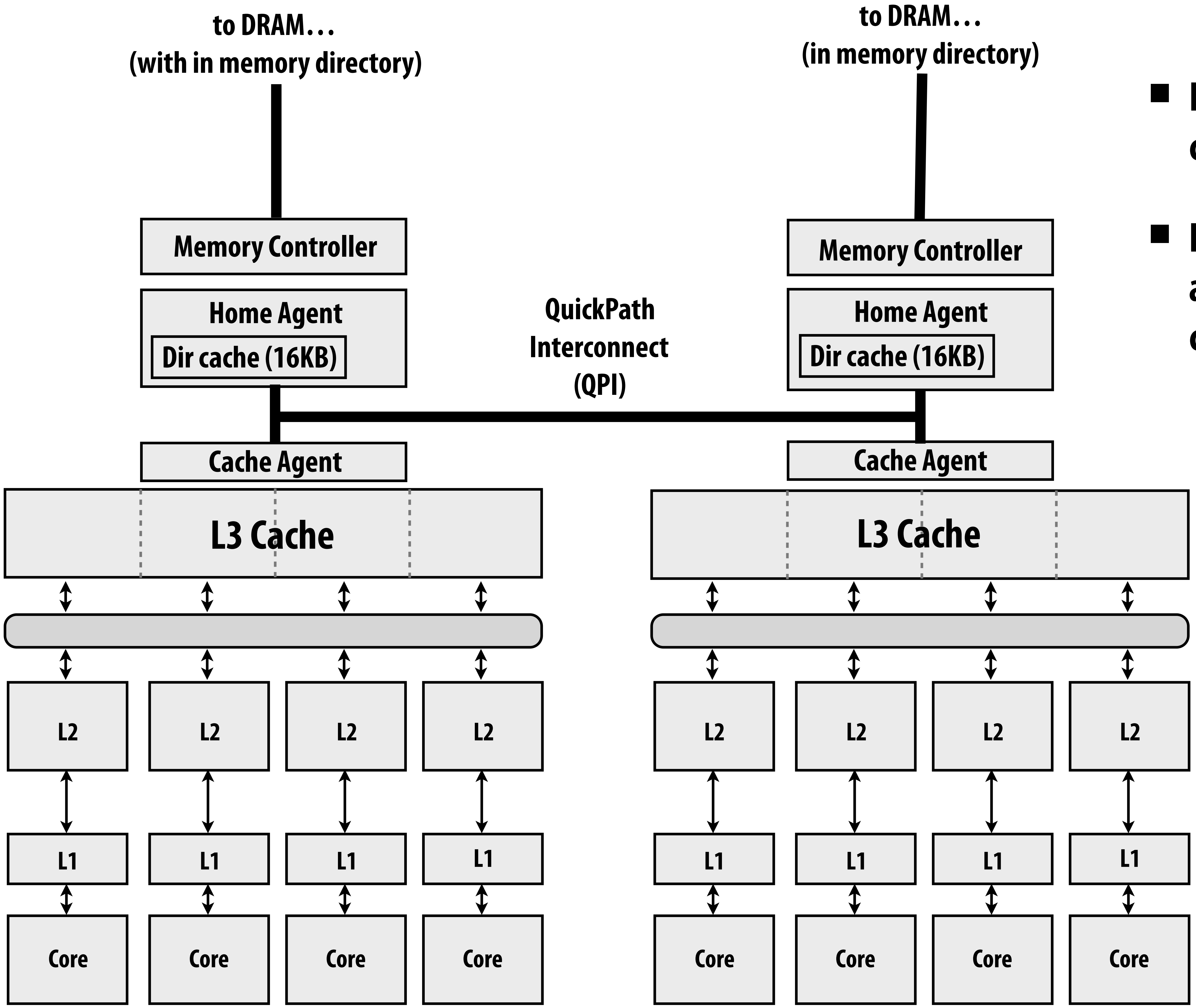
- **Key observation: the majority of memory is NOT resident in cache. And to carry out coherence protocol the system only needs sharing information for lines that are currently in cache**
  - Most directory entries are empty most of the time
  - 1 MB cache, 1 GB memory per node → 99.9% of directory entries are empty

Directory coherence in Intel Core i7 CPU



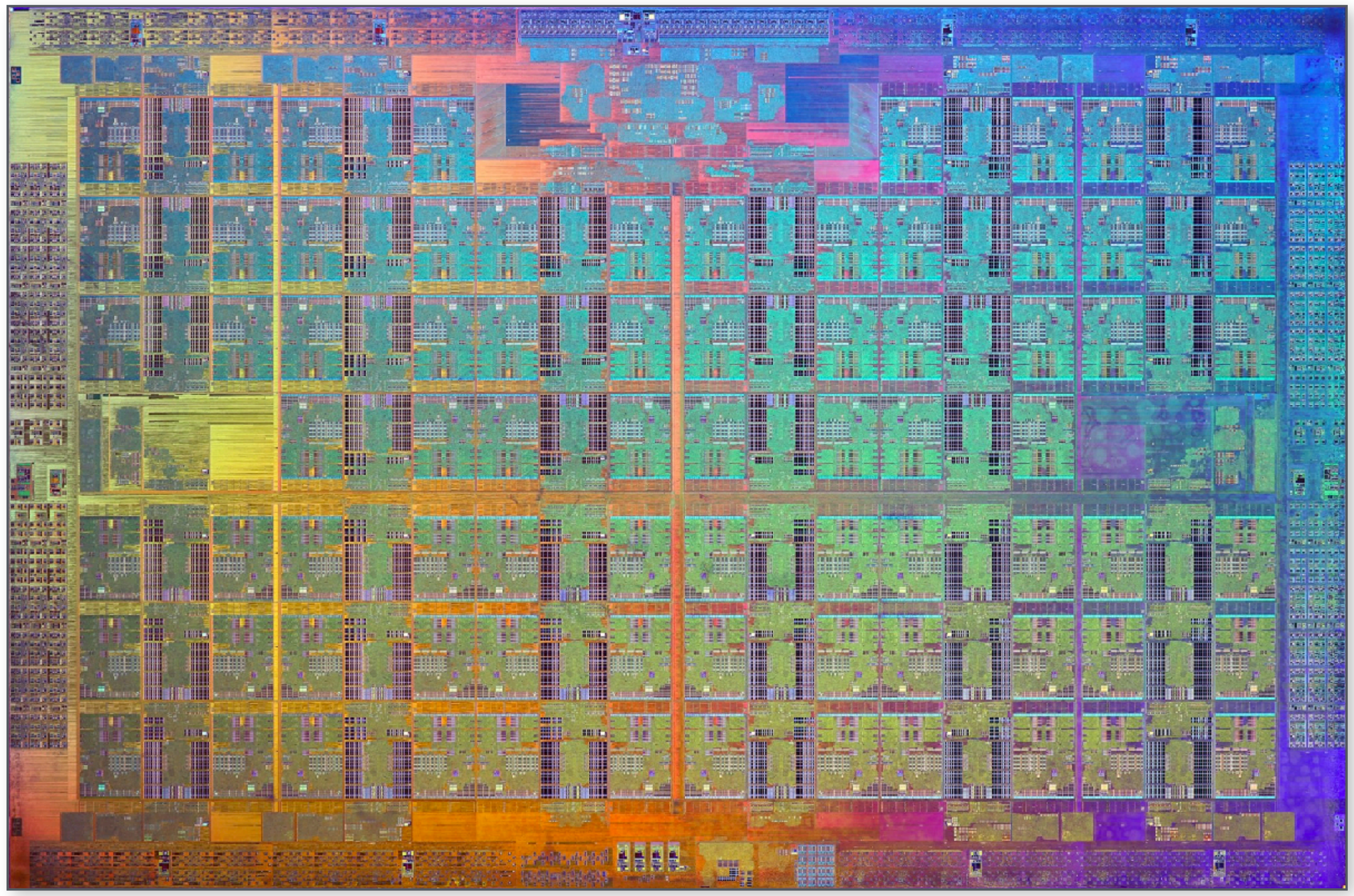
- **L3 serves as centralized directory for all lines in the L3 cache**
  (Since L3 is an inclusive cache, any line in L2 is guaranteed to also be resident in L3)
- **Directory maintains list of L2 caches containing line**
- **Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line**
  (Core i7 interconnect is a ring, it is not a bus)
- **Directory dimensions:**
  - P=4
  - M = number of L3 cache lines

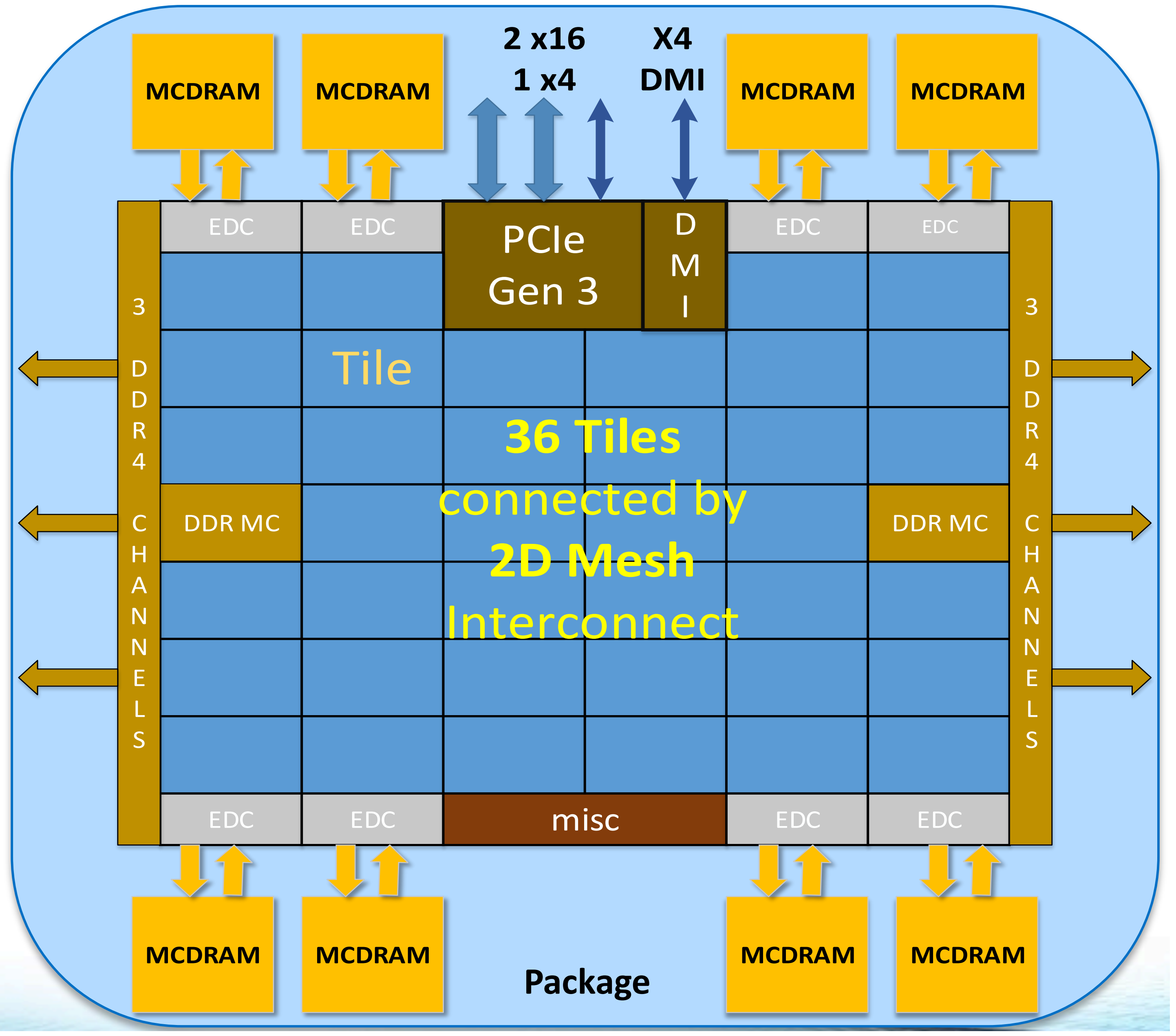
Coherence in multi-socket Intel systems



- L3 directory reduces on-chip coherence traffic (previous slide)
- In-memory directory (cached by home agent/memory controller) reduces coherence traffic between cores

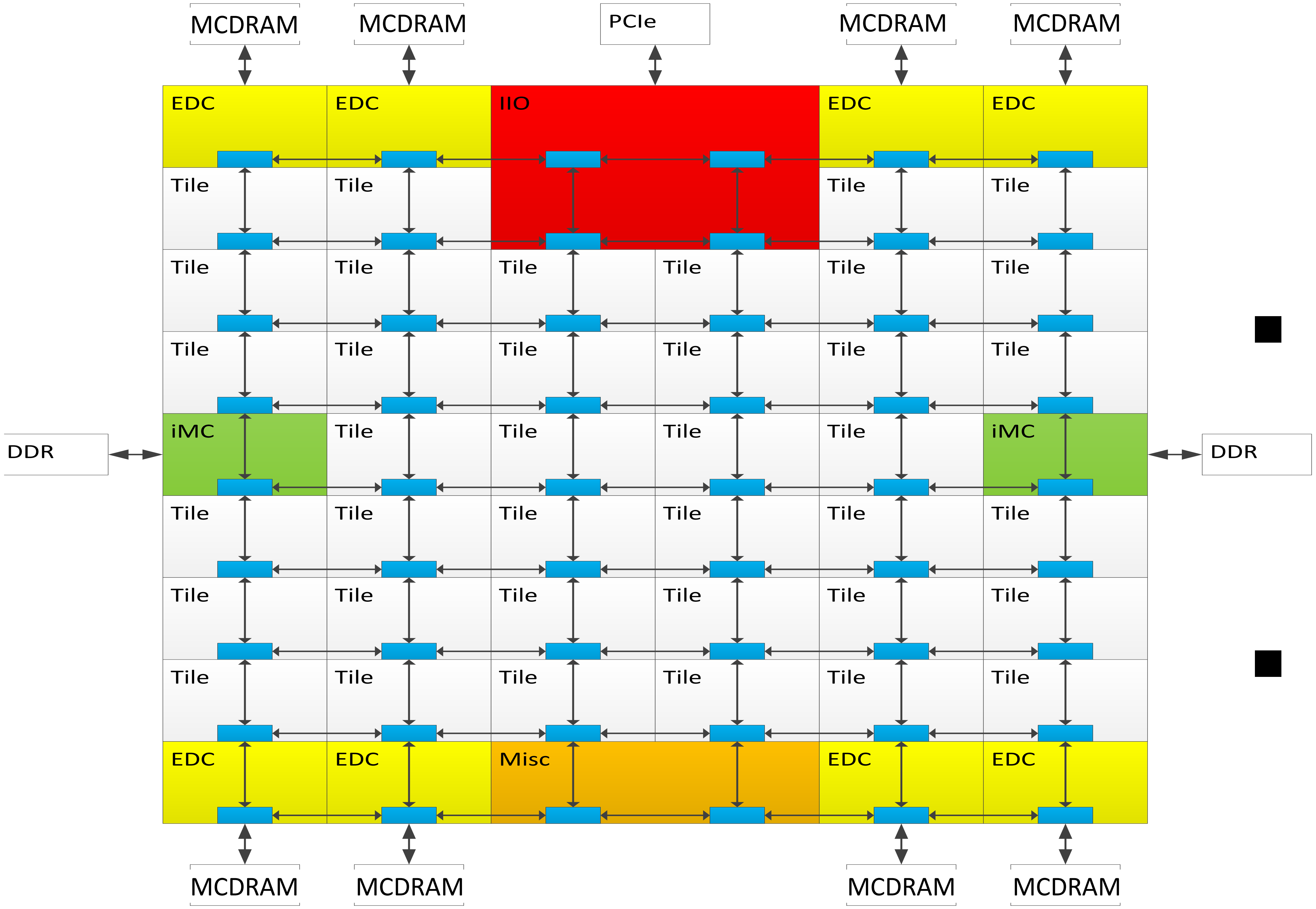
Xeon Phi (Knights Landing) 2015





- "Knights Landing" (KNL)
- 72 cores
  - Two 16-wide SIMD (AVX512) units
  - 4-way multi-threading
- Grouped into 36 tiles
  - 1MB L2 cache per tile
- Peak: 6 TFLOPs (single precision)
- 16 GB of on-package RAM
- Up to 384 GB of of-package DRAM

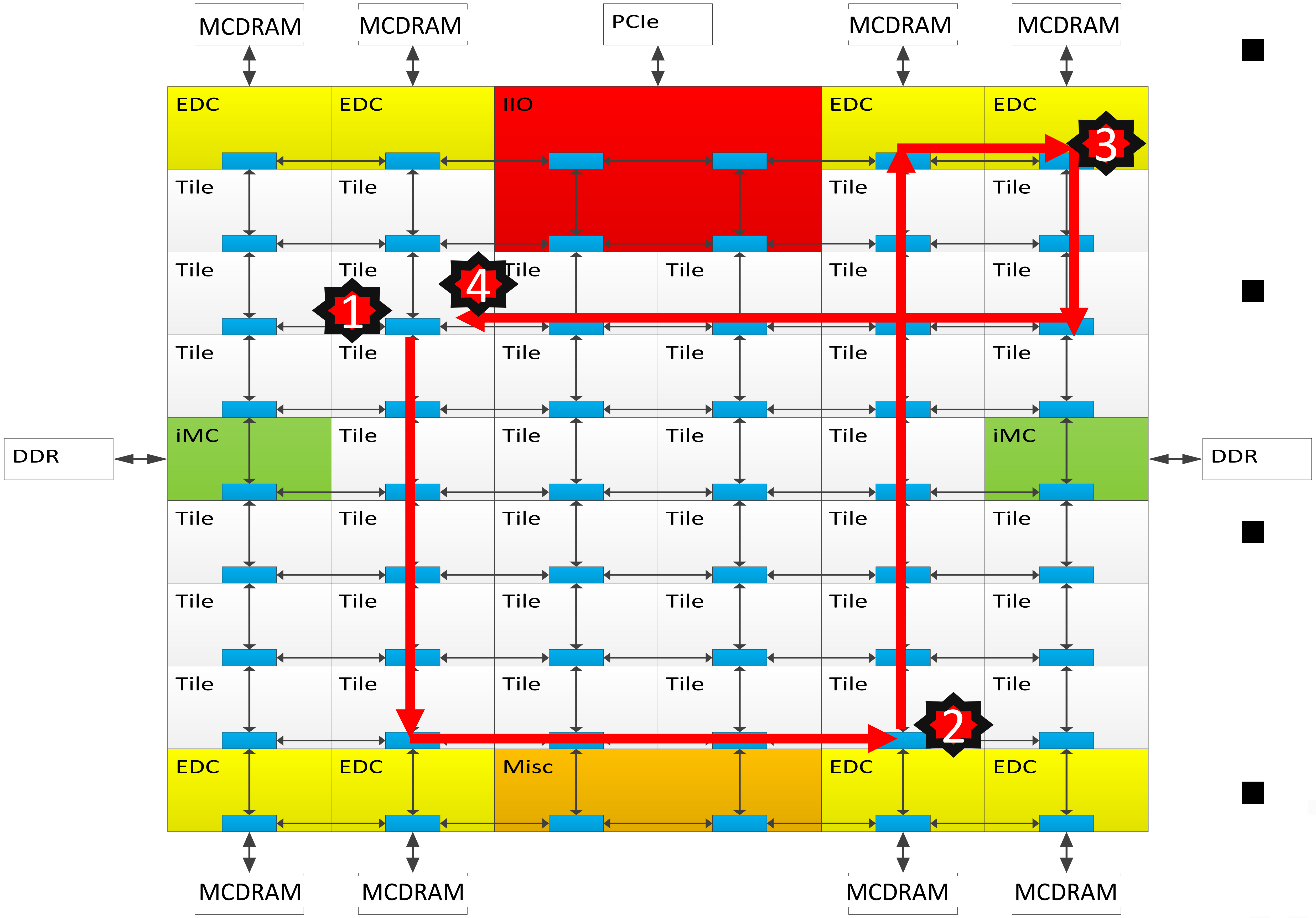
Xeon Phi cache coherence



- Nodes organized as 2D mesh
  - Some nodes are tiles
  - Others are memory interfaces
- X/Y routing to send messages
  - Send horizontally along row first, then vertically
- Directory based scheme for cache coherence

“Slide credit: Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor” A. Sodani (Hot Chips 2015)

Xeon Phi: all-to-all mode directory coherence



- **Directory home tile (node) determined by hash of addresses to tiles**
- **Step 1 (of a memory access): check directory in address' home directory node**
- **Step 2: if miss (line not referenced in directory), must go to memory to retrieve data**
- **Step 3: memory responds directly to requestor**

"Slide credit: Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor" A. Sodani (Hot Chips 2015)

Summary: directory-based coherence

- Primary observation: broadcast doesn't scale, but luckily we don't need broadcast to ensure coherence because often the number of caches containing a copy of a line is small

- Instead of snooping, just store the list of sharers in a "directory" and check the list as necessary

- One challenge: reducing overhead of directory storage
  - Use hierarchies of processors or larger line sizes
  - Limited pointer schemes: exploit fact the most processors not sharing line
  - Exploit fact that most lines are not in cache

Implementing Synchronization

Now that you understand implementations of cache coherence, the cost of implementing synchronization primitives on a modern machine will become very apparent.

Three phases of a synchronization event

1. Acquire method
   - How a thread attempts to gain access to protected resource

2. Waiting algorithm
   - How a thread waits for access to be granted to shared resource

3. Release method
   - How thread enables other threads to gain resource when its work in the synchronized region is complete

Busy waiting

- **Busy waiting (a.k.a. “spinning”)**

```
while (condition X not true) {}
logic that assumes X is true
```

- **In classes like CS107/CS110 or in operating systems, you have certainly also talked about synchronization**
  - **You might have been taught busy-waiting is bad: why?**

“Blocking” synchronization

- **Idea: if progress cannot be made because a resource cannot be acquired, it is desirable to free up execution resources for another thread (preempt the running thread)**

```
if (condition X not true)
   block until true;  // OS scheduler de-schedules thread
                      // (let’s another thread use the processor)
```

- **pthreads mutex example**

```
pthread_mutex_t mutex;
pthread_mutex_lock(&mutex);
```

Busy waiting vs. blocking

- **Busy-waiting can be preferable to blocking if:**
  - Scheduling overhead is larger than expected wait time
  - A processor's resources not needed for other tasks
    - This is often the case in a parallel program since we usually don't oversubscribe a system when running a performance-critical parallel app (e.g., there aren't multiple CPU-intensive programs running at the same time)
    - Clarification: be careful to not confuse the above statement with the value of multi-threading (interleaving execution of multiple threads/tasks to hiding long latency of memory operations) with other work within the same app.

- **Examples:**

```
pthread_spinlock_t spin;          int lock;
pthread_spin_lock(&spin);         OSSpinLockLock(&lock);   // OSX spin lock
```

Implementing Locks

Warm up: a simple, but incorrect, lock

```
 lock:         ld   R0, mem[addr]      // load word into R0
               cmp  R0, #0             // compare R0 to 0
               bnz  lock               // if nonzero jump to top
               st   mem[addr], #1


  unlock:      st   mem[addr], #0      // store 0 to address
```

Problem: data race because LOAD-TEST-STORE is not atomic!

Processor 0 loads address X, observes 0

Processor 1 loads address X, observes 0

Processor 0 writes 1 to address X

Processor 1 writes 1 to address X

Test-and-set based lock

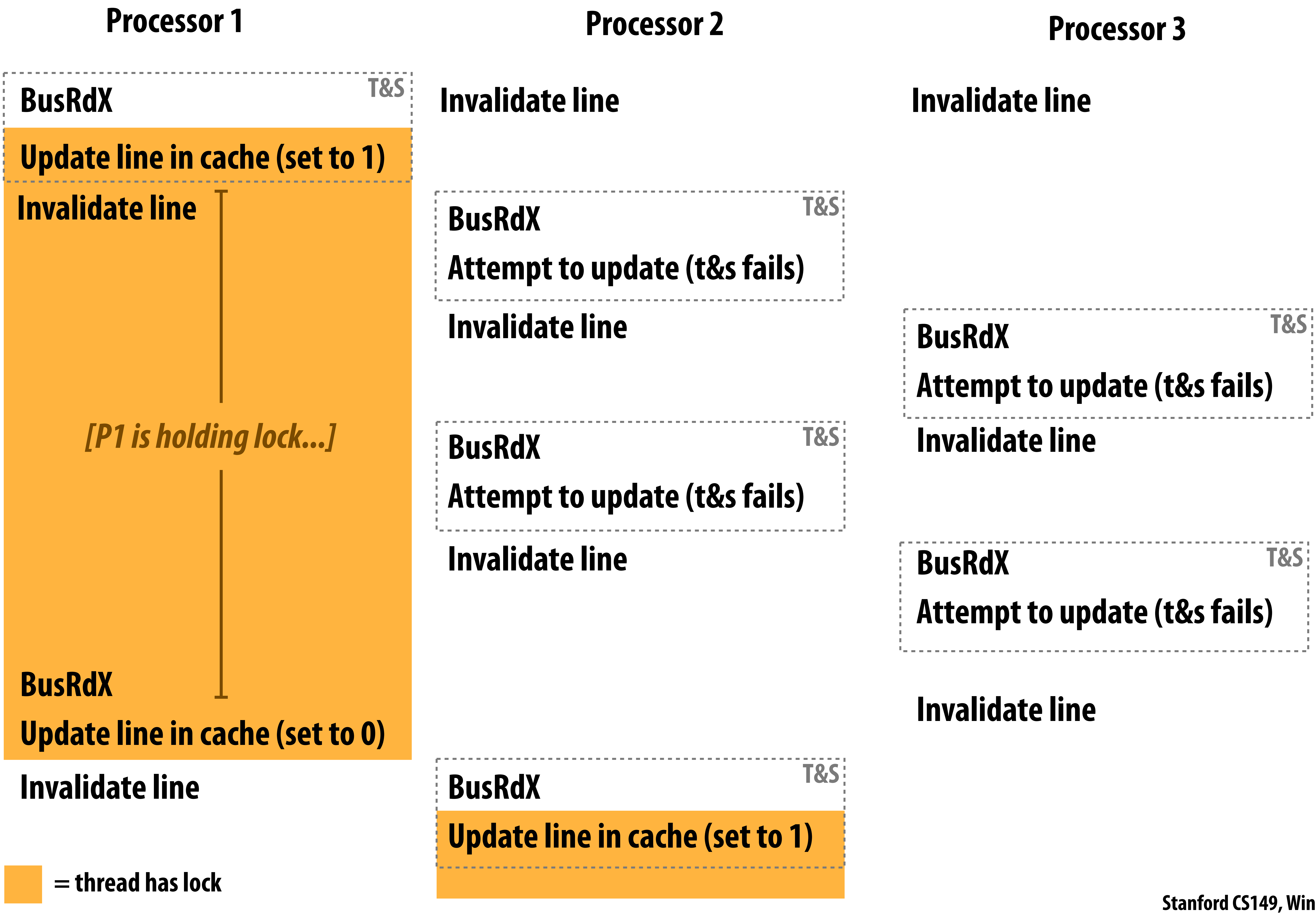
Atomic test-and-set instruction:

```
ts R0, mem[addr]       // load mem[addr] into R0
                       // if mem[addr] is 0, set mem[addr] to 1
```

```
lock:        ts   R0, mem[addr]        // load word into R0
             bnz  R0, lock             // if 0, lock obtained


unlock:      st   mem[addr], #0        // store 0 to address
```

Test-and-set lock: consider coherence traffic



Processor 1

T&S

BusRdX

Update line in cache (set to 1)

Invalidate line

[P1 is holding lock...]

BusRdX

Update line in cache (set to 0)

Invalidate line

Processor 2

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

T&S

BusRdX

Update line in cache (set to 1)

Processor 3

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

🟧 = thread has lock

Check your understanding

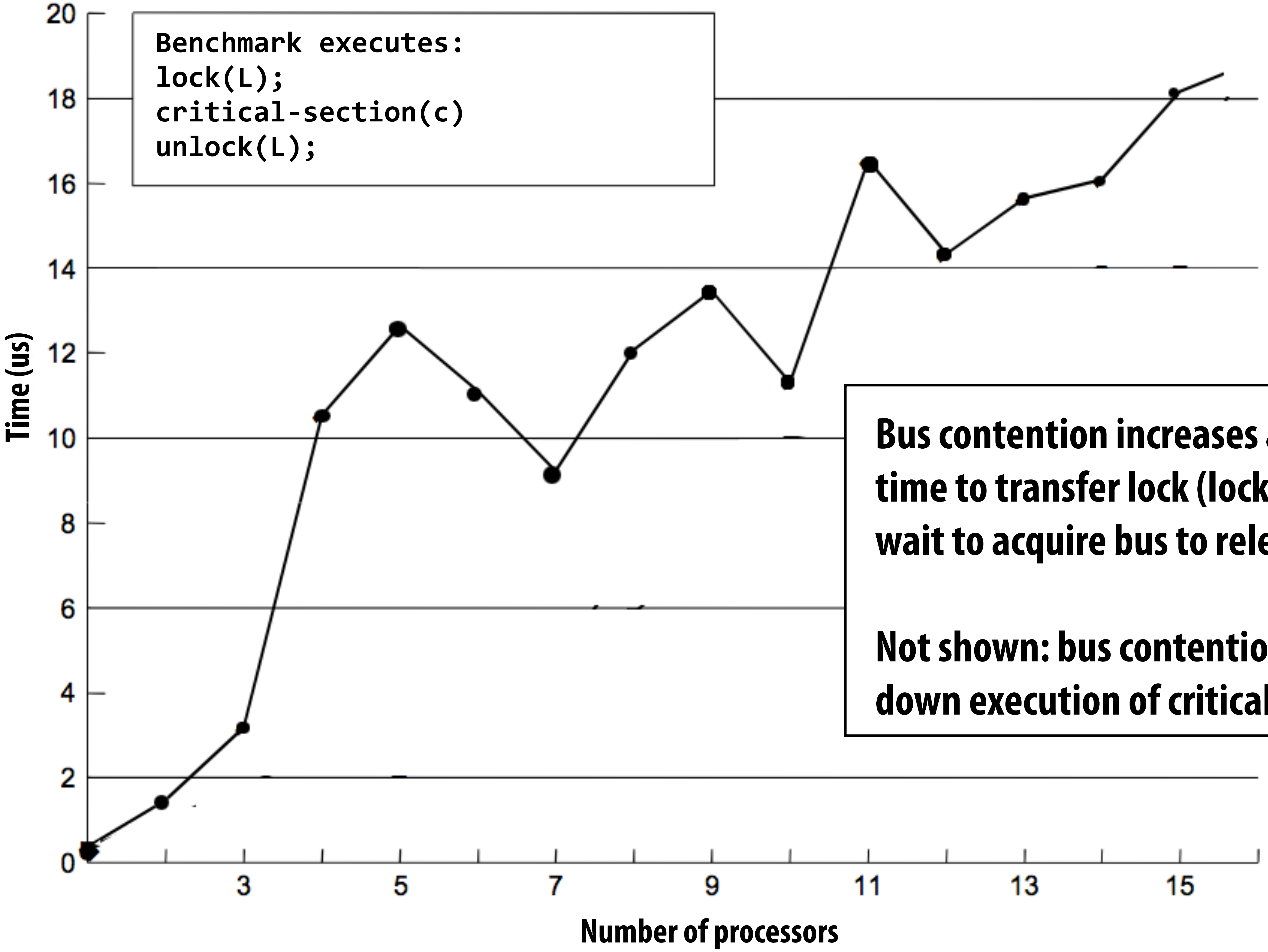
- **On the previous slide, what is the duration of time the thread running on P1 holds the lock?**

- **At what points in time does P1's cache contain a valid copy of the cache line containing the lock variable?**

Test-and-set lock performance

Benchmark: execute a total of N lock/unlock sequences (in aggregate) by P processors

Critical section time removed so graph plots only time acquiring/releasing the lock



```
Benchmark executes:
lock(L);
critical-section(c)
unlock(L);
```

Bus contention increases amount of time to transfer lock (lock holder must wait to acquire bus to release)

Not shown: bus contention also slows down execution of critical section

Figure credit: Culler, Singh, and Gupta

x86 cmpxchg

- **Compare and exchange (atomic when used with lock prefix)**

```
lock cmpxchg dst, src
```

often a memory address

lock prefix (makes operation atomic)

x86 accumulator register

```
if (dst == EAX)
    ZF = 1
    dst = src
else
    ZF = 0
    EAX = dst
```

flag register

Self-check: Can you implement assembly for atomic compare-and-swap using `cmpxchg`?

```
bool compare_and_swap(int* x, int a, int b) {
   if (*x == a) {
     *x = b;
     return true;
   }

   return false;
}
```

Desirable lock performance characteristics

- Low latency
  - If lock is free and no other processors are trying to acquire it, a processor should be able to acquire the lock quickly
- Low interconnect traffic
  - If all processors are trying to acquire lock at once, they should acquire the lock in succession with as little traffic as possible
- Scalability
  - Latency / traffic should scale reasonably with number of processors
- Low storage cost
- Fairness
  - Avoid starvation or substantial unfairness
  - One ideal: processors should acquire lock in the order they request access to it

Simple test-and-set lock: low latency (under low contention), high traffic, poor scaling, low storage cost (one int), no provisions for fairness

Test-and-test-and-set lock

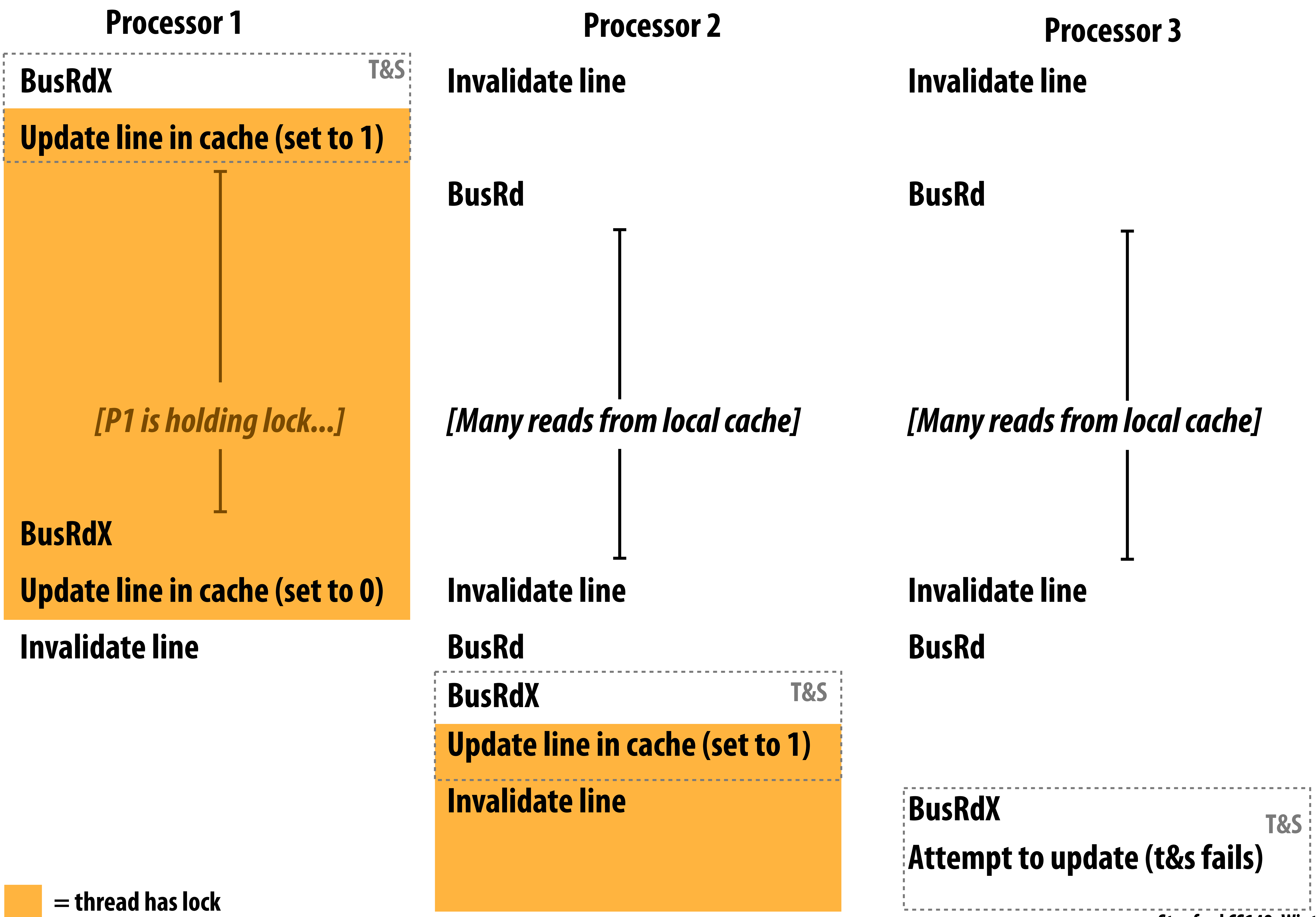
```
void Lock(int* lock) {
  while (1) {

    while (*lock != 0);           // while another processor has the lock…
                                  // (assume *lock is NOT register allocated)


    if (test_and_set(*lock) == 0) // when lock is released, try to acquire it
      return;
  }
}

void Unlock(int* lock) {
   *lock = 0;
}
```

Test-and-test-and-set lock: coherence traffic



Processor 1	Processor 2	Processor 3
BusRdX T&S	Invalidate line	Invalidate line
Update line in cache (set to 1)		
	BusRd	BusRd
[P1 is holding lock...]	*[Many reads from local cache]*	*[Many reads from local cache]*
BusRdX		
Update line in cache (set to 0)	Invalidate line	Invalidate line
Invalidate line	BusRd	BusRd
	BusRdX T&S	
	Update line in cache (set to 1)	
	Invalidate line	BusRdX T&S
		Attempt to update (t&s fails)

🟧 = thread has lock

Test-and-test-and-set characteristics

- **Slightly higher latency than test-and-set in uncontended case**
  - Must test... then test-and-set
- **Generates much less interconnect traffic**
  - One invalidation, per waiting processor, per lock release (O(P) invalidations)
  - This is $O(P^2)$ interconnect traffic if all processors have the lock cached
  - Recall: test-and-set lock generated one invalidation per waiting processor per test
- **More scalable (due to less traffic)**
- **Storage cost unchanged (one int)**
- **Still no provisions for fairness**

Test-and-set lock with back off

Upon failure to acquire lock, delay for awhile before retrying

```
void Lock(volatile int* l) {
  int amount = 1;
  while (1) {
    if (test_and_set(*l) == 0)
      return;
    delay(amount);
    amount *= 2;
  }
}
```

- Same <u>uncontended</u> latency as test-and-set, but potentially higher latency under contention. Why?
- Generates less traffic than test-and-set (not continually attempting to acquire lock)
- Improves scalability (due to less traffic)
- Storage cost unchanged (still one int for lock)
- Exponential back-off can cause severe unfairness
  - Newer requesters back off for shorter intervals

Ticket lock

Main problem with test-and-set style locks: upon release, all waiting processors attempt to acquire lock using test-and-set

```
struct lock {
   int next_ticket;
   int now_serving;
};

void Lock(lock* l) {
  int my_ticket = atomic_increment(&l->next_ticket);   // take a “ticket”
  while (my_ticket != l->now_serving);                 // wait for number
}                                                      // to be called

void unlock(lock* l) {
  l->now_serving++;
}
```

No atomic operation needed to acquire the lock (only a read)

Result: only one invalidation per lock release (O(P) interconnect traffic)

Array-based lock

Each processor spins on a different memory address

Utilizes atomic operation to assign address on attempt to acquire

```
struct lock {
   padded_int status[P];    // padded to keep off same cache line
   int head;
};

int my_element;

void Lock(lock* l) {
  my_element = atomic_circ_increment(&l->head);    // assume circular increment
  while (l->status[my_element] == 1);
}

void unlock(lock* l) {
  l->status[my_element] = 1;
  l->status[circ_next(my_element)] = 0;            // next() gives next index
}
```

O(1) interconnect traffic per release, but lock requires space linear in P

Also, the atomic circular increment is a more complex operation (higher overhead)

Additional atomic operations

Atomic operations provided by CUDA

```
int   atomicAdd(int* address, int val);
float atomicAdd(float* address, float val);
int   atomicSub(int* address, int val);
int   atomicExch(int* address, int val);
float atomicExch(float* address, float val);
int   atomicMin(int* address, int val);
int   atomicMax(int* address, int val);
unsigned int atomicInc(unsigned int* address, unsigned int val);
unsigned int atomicDec(unsigned int* address, unsigned int val);
int   atomicCAS(int* address, int compare, int val);
int   atomicAnd(int* address, int val);  // bitwise
int   atomicOr(int* address, int val);   // bitwise
int   atomicXor(int* address, int val);  // bitwise
```

(omitting additional 64 bit and unsigned int versions)

Implementing atomic fetch-and-op

```
// atomicCAS:
// atomic compare and swap performs the following logic atomically
int atomicCAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
```

Exercise: how can you build an atomic fetch+op out of atomicCAS()?

Example: atomic_min()

```
   int atomic_min(int* addr, int x) {
      int old = *addr;
      int new = min(old, x);
      while (atomicCAS(addr, old, new) != old) {
        old = *addr;
        new = min(old, x);
      }
   }
```

What about these operations?

```
int  atomic_increment(int* addr, int x);   // for signed values of x
void lock(int* addr);
```

Load-linked, store conditional (LL/SC)

- Pair of corresponding instructions (not a single atomic instruction like compare-and-swap)
  - load_linked(x): load value from address
  - store_conditional(x, value): store value to x, if x hasn't been written to since corresponding LL

- Corresponding ARM instructions: LDREX and STREX

- How might LL/SC be implemented on a cache coherent processor?

C++ 11 atomic<T>

- Provides atomic read, write, read-modify-write of entire objects
  - Atomicity may be implemented by mutex or efficiently by processor-supported atomic instructions (if T is a basic type)

- Provides memory ordering semantics for operations before and after atomic operations
  - By default: sequential consistency
  - See std::memory_order or more detail

```
atomic<int> i;
i++; // atomically increment i

int a = i;
// do stuff
i.compare_exchange_strong(a, 10);   // if i has same value as a, set i to 10
bool b = i.is_lock_free();          // true if implementation of atomicity
                                    // is lock free
```

- Will be useful if implementing the ideas in our future lock-free programming lecture

Implementing Barriers

Implementing a centralized barrier

(Barrier for P processors, based on shared counter)

```
struct Barrier_t {
  LOCK lock;
  int counter;              // initialize to 0
  int flag;
};

// parameter p gives number of processors that should hit the barrier
void Barrier(Barrier_t* b, int p) {
  lock(b->lock);
  if (b->counter == 0) {
    b->flag = 0;            // first thread arriving at barrier clears flag
  }
  int num_arrived = ++(b->counter);
  unlock(b->lock);

  if (num_arrived == p) {  // last arriver sets flag
    b->counter = 0;
    b->flag = 1;
  }
  else {
    while (b->flag == 0);  // wait for flag
  }
}
```

Does it work? Consider:

```
do stuff ...
Barrier(b, P);
do more stuff ...
Barrier(b, P);
```

Correct centralized barrier

```
struct Barrier_t {
  LOCK lock;
  int arrive_counter;   // initialize to 0 (number of threads that have arrived)
  int leave_counter;    // initialize to P (number of threads that have left barrier)
  int flag;
};

void Barrier(Barrier_t* b, int p) {
  lock(b->lock);
  if (b->arrive_counter == 0) {       // if first to arrive...
    if (b->leave_counter == P) {      // check to make sure no other threads “still in barrier”
       b->flag = 0;                   // first arriving thread clears flag
    } else {
      unlock(lock);
      while (b->leave_counter != P);  // wait for all threads to leave before clearing
      lock(lock);
      b->flag = 0;                    // first arriving thread clears flag
    }
  }
  int num_arrived = ++(b->arrive_counter);
  unlock(b->lock);

  if (num_arrived == p) {     // last arriver sets flag
    b->arrive_counter = 0;
    b->leave_counter = 1;
    b->flag = 1;
  }
  else {
    while (b->flag == 0);     // wait for flag
    lock(b->lock);
    b->leave_counter++;
    unlock(b->lock);
  }
```

Main idea: wait for all processes to leave first barrier, before clearing flag for entry into the second

Centralized barrier with sense reversal

```
struct Barrier_t {
  LOCK lock;
  int  counter;               // initialize to 0
  int  flag;                  // initialize to 0
};

int private_sense = 0;        // private per processor. Main idea: processors wait
                              // for flag to be equal to private_sense

void Barrier(Barrier_t* b, int p) {
  private_sense = (private_sense == 0) ? 1 : 0;
  lock(b->lock);
  int num_arrived = ++(b->counter);
  if (b->counter == p) {      // last arriver sets flag
    unlock(b->lock);
    b->counter = 0;
    b->flag = private_sense;
  }
  else {
    unlock(b->lock);
    while (b.flag != private_sense);  // wait for flag
  }
```

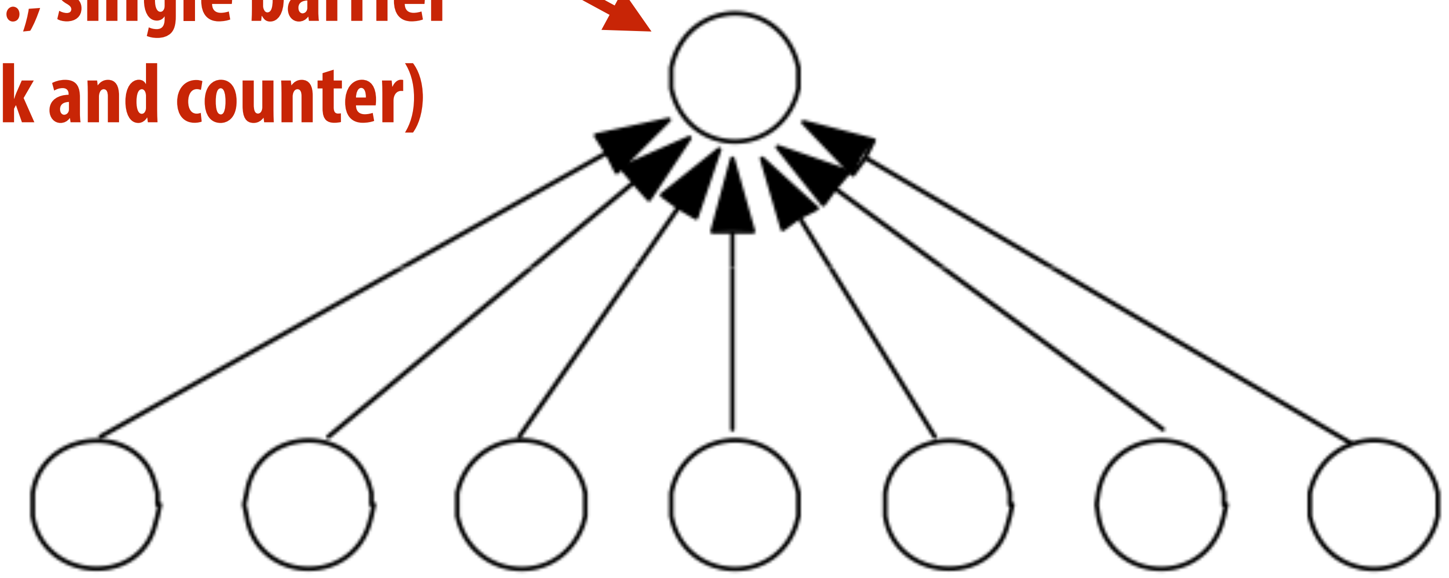
Sense reversal optimization results in one spin instead of two

Centralized barrier: traffic

- **O(P) traffic on interconnect per barrier:**
  - All threads: 2P write transactions to obtain barrier lock and update counter (O(P) traffic assuming lock acquisition is implemented in O(1) manner)
  - Last thread: 2 write transactions to write to the flag and reset the counter (O(P) traffic since there are many sharers of the flag)
  - P-1 transactions to read updated flag

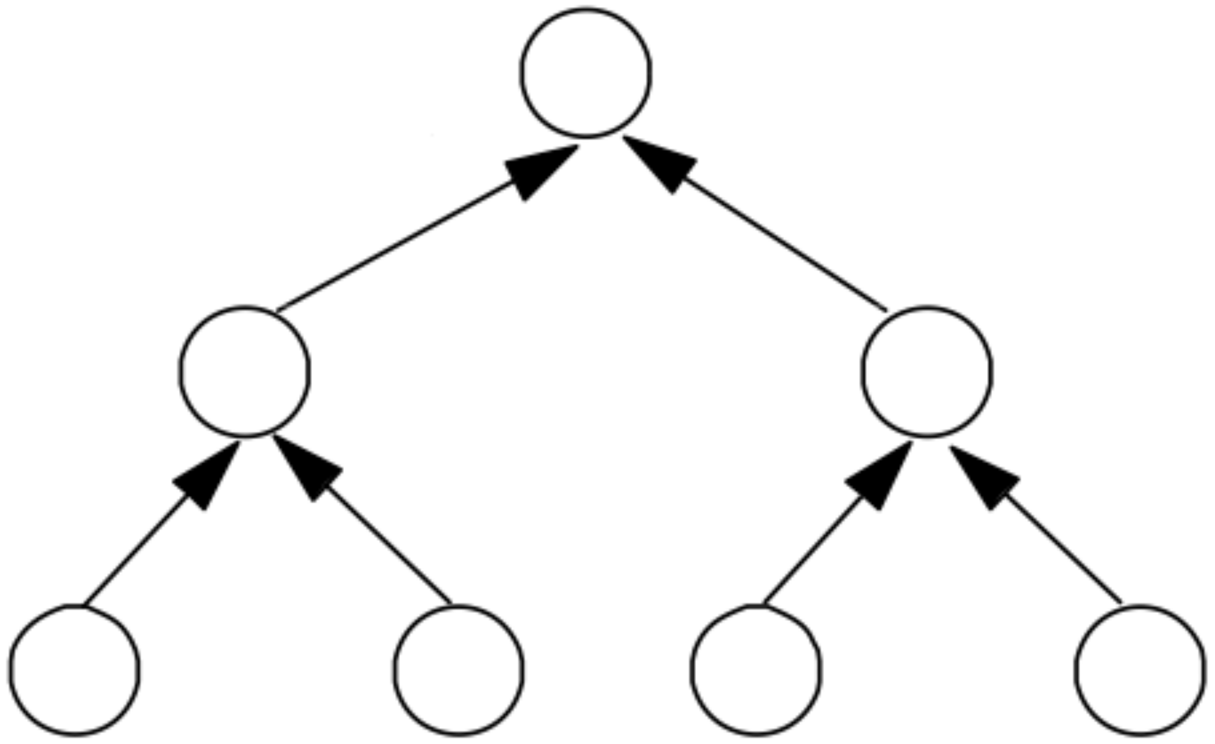
- **But there is still serialization on a single shared lock**
  - So span (latency) of entire operation is O(P)
  - Can we do better?

Combining tree implementation of barrier



High contention!
(e.g., single barrier
lock and counter)

Centralized Barrier



Combining Tree Barrier

- Combining trees make better use of parallelism in more complex interconnect topologies
  - lg(P) span (latency)
- Barrier acquire: when processor arrives at barrier, performs increment of parent counter
  - Process recurses to root
- Barrier release: beginning from root, notify children of release

Coming up…

- Imagine you have a shared variable for which contention is low. So it is unlikely that two processors will enter the critical section at the same time?

- You could hope for the best, and avoid the overhead of taking the lock since it is likely that mechanisms for ensuring mutual exclusion are not needed for correctness
  - Take a "optimize-for-the-common-case" attitude

- What happens if you take this approach and you're wrong: in the middle of the critical region, another process enters the same region?

Preview: transactional memory

```
atomic
{   // begin transaction

   perform atomic computation here ...

}   // end transaction
```

Instead of ensuring mutual exclusion via locks, system will proceed as if no synchronization was necessary. (it speculates!)

System provides hardware/software support for “rolling back” all loads and stores in the critical region if it detects (at run-time) that another thread has entered same region at the same time.